University of Salerno

Department of Information and Electrical Engineering and Applied Mathematics

Master Degree in Computer Science

Embedded Digital Controllers

A.A. 2023/2024

Group 4:
Iannaccone Martina, mat.: 0622702102
Morrone Antonio Alessandro, mat.: 0622702112
Pepe Lorenzo, mat.: 0622702121
Vitolo Miriam, mat.: 0622702097

Contents

1 A brief introduction to Model Based Development **3**
1.1 MBD Principles . . . 3
1.2 MBD steps . . . 4

2 Motors characterization and PID design **5**
2.1 Electric Motor Characterization . . . 5
2.2 PI controller design . . . 10

3 Model design and MIL validation **14**
3.1 Model development . . . 14
3.2 MIL Validation . . . 16

4 Model discretization and SIL validation **19**
4.1 Discretization process . . . 19
4.2 SIL validation . . . 21

5 PIL validation **23**
5.1 Consideration on SIL and PIL validations . . . 25

6 Switch model design **26**
6.1 MIL validation of the improved model . . . 27

7 Hardware description **31**
7.1 Motor driver setup . . . 32
7.2 STM32 project setup . . . 32
7.3 Clock tree setup . . . 34

8 Software implementation **36**
8.1 Lynxmotion controller driver . . . 36
8.2 Encoder driver . . . 38
8.3 PID controller driver . . . 40
8.4 Logging driver . . . 43
8.5 Callback function . . . 44

9 Validation on the real system **45**
9.1 System without controllers . . . 46
9.2 System with controlled motor . . . 47
9.3 System with controlled motor and axle control . . . 48

Introduction

This report summaries the work done for the examination of “Embedded Digital Controllers”, held by prof. Basile Francesco, for the master’s degree course in Computer Science, specialization in Embedded Systems, of the Department of Information and Electrical Engineering and Applied Mathematics, in the academic year 2023/2024. The rover system consists of several components:

- an "STM32F401RE" microcontroller;
- a PS2/PSX BT controller "Lynxmotion", used to navigate the rover, with it’s bluetooth receiver;
- two motor driver "Sabertooth 2x12";
- four electric motor "36GP540-51";

Other devices were present on the system but being not fundamental for the purpose of this project, they were removed. Too many devices can introduce computational delays into the system, which are not ideal for control systems in general. The microcontroller will command the electric motor using a Pulse Width Modulation (PWM) signal, directed into the motor controllers, which will use a voltage control to deliver the right amount of voltage (between 0V and 12V) to each motor.

The project will follow the Model Based Development (MBD) method, covering the first three step of the process: Model In the Loop (MIL), Software In the Loop (SIL) and Processor In the Loop (PIL). The fourth and last part of the MDB method, the Hardware In the Loop (HIL), will not be covered in this project.

Lastly, the project will be developed using software such as MATLAB, Simulink, Control System Design (Sisotool) and STM32 CubeIDE.

1. A brief introduction to Model Based Development

The Model Based Developement, or MBD in short, part of the MBSE (Model Based Software Engineering), is an approach to the development of complex systems that have become very complicated to develop, test and certify; particularly used in sectors such as the automotive, aerospace and electronics industries. The MBD makes it possible to design, simulate, verify and implement systems from mathematical models rather than through the traditional approach based solely on code or circuit diagrams. Having separate process, and testing at every step, help develop strong and resilient systems in relatively small time. The process, as already stated, is divided into four parts:

- MIL: Model in the Loop;
- SIL: Software in the Loop;
- PIL: Processor in the Loop;
- HIL: Hardware in the Loop.

Before delving into explaining the MBD steps, first a brief explanation of the principles of the process, with some pros and cons.

1.1 MBD Principles

In the MBD the model represents the core of the development cycle. All phases are based on a mathematical or graphical model, describing the behaviour of the system. In addition, with tools like Matlab and Simulink, which allow to quickly and simply modify the model, it's possible to generate the code automatically and do testing, both in virtualised environments and on the target hardware, modifying and adjusting the systems has become an easier job. Every step of the process bring several advantages:

- having a block model/design in a virtualised environment, makes changes easy and painless;
- tools like Matlab make uses of the powerful computational capacities of now days computers, to simplify the necessary mathematical calculations;
- Simulink bring the project to the next level, allowing to simulate and validate the model through the first three phases on it's own;
- other Matlab plugins, used in this process, makes the project flow faster and easier; in particular tools like "System Identification Toolbox" and "Control System Designer" were essential for the project.

This approach bring indeed more pros than cons: it speed up designing, simulation and corrections, if any, to the model, even further into the project.

1.2 MBD steps

A basic model must represent a physical device through a mathematical function, usually a transfer function. If this function is not known, a suitable function, reflecting as closely as possible the physical device we wish to simulate, must be calculated and tested before developing the model. There are a few methods of computing such functions: for our purposes the step response method has been used and will be explained in details in the next chapters.

When the mathematics model of the devices are known, the preliminary step before the MBD process is the creation of a model, representing the system: in our case the model is a rover, equipped with four electric DC motor, each with an rotational encoder, controlled by an STM microcontroller and guided via a BT controller. In our specific case the system, replicated in Matlab, is simply all the pairs controller-motor, with two addition PID controller for balancing the speed on each axis.

After developing the model, the next step is the MIL validation: the system must be tested in different ways. First the response of the system is tested using a step- response method; then, adding some form of disturbs on the chain of one, or more, motors, the robustness of the system is tested, checking whether it can return to a steady state in a reasonable time, without fluctuations. If the system meets the requirements, the project can proceed to the second step.

The Software in the Loop tests whether the code, generated automatically by Matlab, is correct and behave as expected in a simulated environment. This process require the redefinition of the model, from a continuous time model to a discrete time model. Thanks to Matlab this step can be performed quickly. The process will be described in detail in the following chapters. If the new discrete model generates a code that behave properly, as intended, the SIL part is validated and the project can pass onto the next step.

Processor in the Loop bring the project into the hardware: it aims to control whether the target hardware, accountable to host the code, have enough computation power to withstand the generated program. It's a key step in validating embedded systems. The PIL, use the generated code from the SIL to simulate the process: the process is still being simulated, but this time using the target processor, designated to be used on the final system. This part compute the execution time, the computational load and the resource management of the processor chosen. Validating the PIL step, means that the hardware can indeed withstand the computation needed for the generated code, thus validating the hardware too.

The last step is the Hardware in the Loop: the model is still simulated but with a high degree of fidelity relative to the final system. The main difference between PIL and HIL lies in the testing of all the sensors, actuators and physical interfaces. In fact, the target processor is connected to a high performance computer, that act as the real model being tested (in our case the electric motors). Exploiting this high performance computer, like Speedgoat, makes possible to tests all the aspect of the system at this point of the project, before loading the code into the real hardware, making possible to further check for possible flaws in the project.

However, in our project, the HIL part is an overshoot and will not be used.

2. Motors characterization and PID design

In this chapter will be described the process of computing an accurate transfer function for each motor and subsequent PID design. The tools used for these purposes are:

- Matlab: saving the data in output from all the other tools;
- Simulink: for obtaining data from the encoder, for each motor, when applied a step-like tension;
- System Identification Toolbox: for computing the transfer functions of each motor, from the data acquired from Simulink;
- Control System Designer: for designing PID controllers, relative to each motor, using the embedded "PID Tuner".

These process will be described in the next paragraph.

2.1 Electric Motor Characterization

The first step of the project is the characterization of the electric motor: without a proper transfer function, the project cannot start and the model cannot be developed. The motor in our possession are the "36GP540-51" and it's datasheet, with the associate datasheet of the associated rotary encoder, can be seen in **Table 2.1** e **Table 2.2**.

	No Load	**Nominal torque**	**At stall**
Model	36GP540-51		
Voltage	12V		
Total gear ratio	1:51		
Speed	170 RPM ±10%	145 RPM ±10%	0 RPM
Current	$\leq 0,5A$	$\leq 3A$	$\leq 20A$
Torque	–	11,5 Kg/cm	78 Kg/cm

Table 1: Motor "36GP540-51" datasheet

Voltage	5V
Signals	Square wave AB 90°
Temperature	$[-20, 80]$ °C
Pulse	12 PPR
Frequency	800 KHz

Table 2: Encoder datasheet

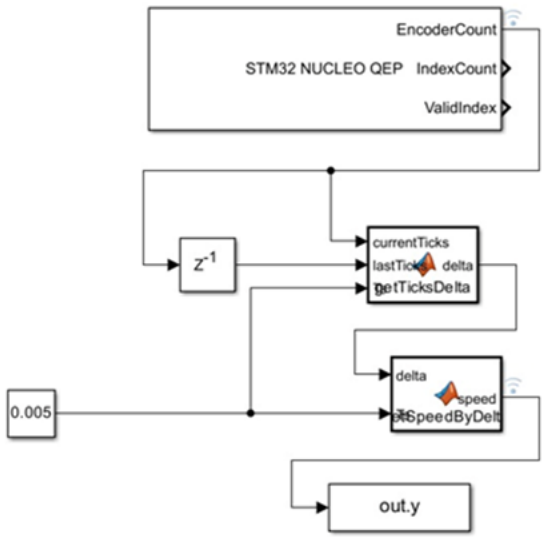
The color code of the cables for the motor are as follows:

- Yellow-White: delivers voltage to the motor;
- Black: ground for the encoder;
- Red: deliver power to the encoder;
- Blue-Green: channel A and B of the encoder, delivering the square wave signals.

The characterization of the motors was done by implementing a model in "Simulink" and using the step response method by collecting the output of the system. The method of the step response consists of applying an input u that suddenly changes from an initial value (often 0) to a final value (in this case, 6V which is half of the maximum voltage). Step response is one of the most widely used methods for characterizing dynamic systems because it provides important information, such as:

- rise time: how quickly the system reacts to the input change;
- overshoot: whether the system "overshoots" its target position before stabilizing;
- settling time: how long it takes the system to stabilize after the step;
- stability: whether the system is stable or tends to oscillate continuously.

A model of the encoder has been developed in Simulink to acquire the RPM in response to a 6V step stimulus. For this particular part of the project, the bench-top generator "RS-PRO RS3005D", supplied to us by the DIEM, has been used to deliver the exact 6V stimulus to the motors. In **Figure 1a** the Simulink model for the encoder has been reported, while in **Figure 1b** it's reported the simple for loop used in Matlab to generate the input values array that we need, paired with the output values array, to compute the transfer function.




```
for ii = 1:length(y)
    if y(ii) ~= 0
        u(ii) = 6;
    else
        u(ii) = 0;
    end
end
```

Figure 1: a) Simulink encoder design - b) for-loop used in Matlab

The Simulink design make use of the "STM32 Nucleo QEP" block, provided during the course relative to this examination, to acquire the count of the encoder. Then, two Matlab function are accounted for making the calculations necessary to compute the RPM speed:

- "ticksByDelta" compute the difference between the current tick count and the last tick count;
- "SpeedByDelta" use the differences computed by the first function to count the RPM speed.

This last function needs a parameter, computed using the parameters of the motor. This constant is the number of Ticks necessary to count for the rotor to complete a complete revolution. Using the equation:

$$nTicks = PPR \times (\#ofFronts) \times gearRatio = 12 \times 4 \times 51 = 2448.$$

it is possible to compute such number. The number of fronts refers to the square waves fronts to count.

The output of the encoder design is then outputted in the Matlab workspace, through the block "out.y". Then, scanning the just computed output, we use the for-loop visible in **Figure 1b**, to compute an input array of values. Using this scheme it was possible to acquire a pair of arrays values input/output for each motor, used in the tool "System Identification Toolbox" to compute the relative transfer functions.

Now it's possible to use the aforementioned tool to compute the $P(s)$, the transfer functions for our process. In **Figure 2** it's possible to see the toolbox interface.

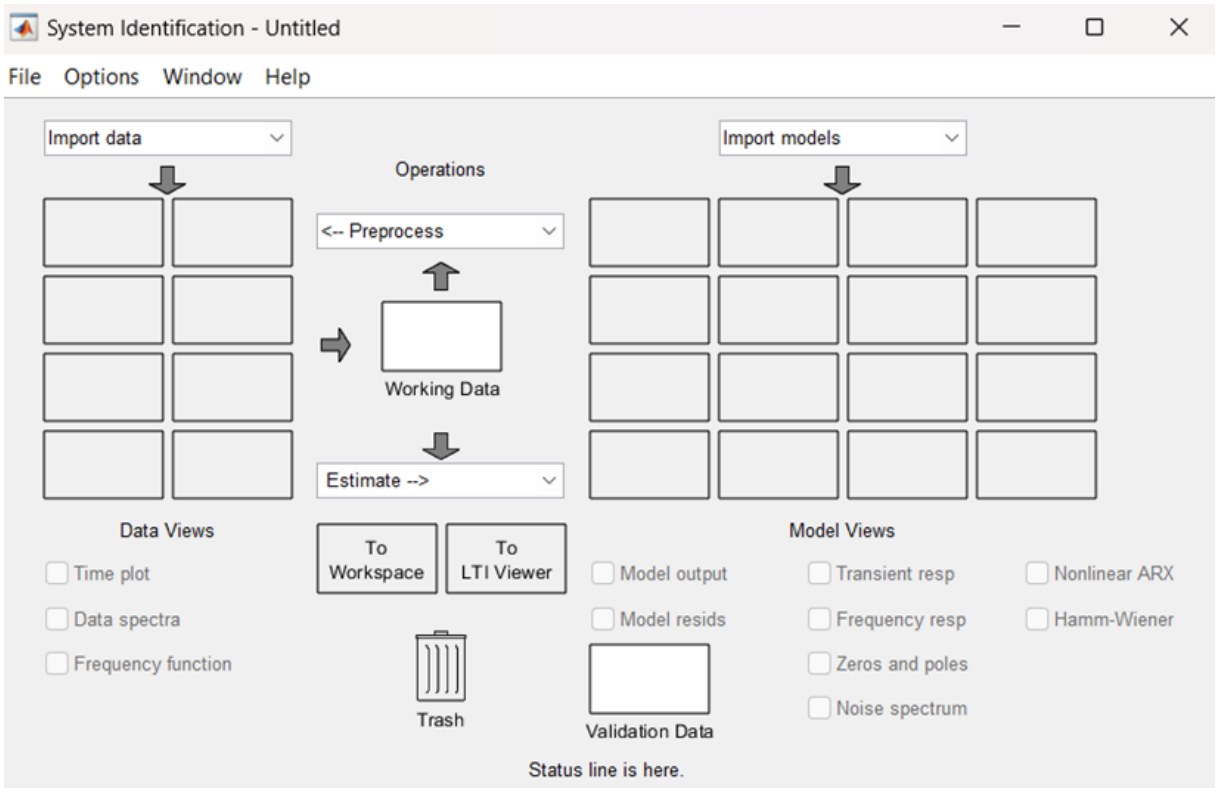


Figure 2: System Identification Toolbox interface

The tools is easy to operate: the first step is to import the input/output arrays in the imported data, as *DataObject*. When importing these data we must specify the start/end time and the sampling time (in our case of 5ms).After importing the data it's possible to estimate the process of the electric motor. To do this it's necessary to set the working data and validation data equal to the imported data (by dragging and dropping the data just imported). Now the toolbox is ready to compute a model: on the "estimate" drop down menu it's possible to select "Transfer Function Model"; this will open another interface when it's possible to select how many poles and zeroes our model should have. We set these parameters so we can have a first order function (one poles and zero zeroes). After the computation it's done, just drag and drop the estimate model to the "To Workspace" box to export it in MATLAB.

The reason for choosing a first order transfer function are mainly two:

- most electric DC motor can be approximated by a first-order model when analysing the relationship between the applied voltage and the output RPM;
- it's easy to evaluate time response, stability and system performances;
- exploiting the first point, we can design a simpler controller in the form of a PI (Proportional-Integral), avoid using the Derivative part.

Since the derivative part do not take action in controlling first order transfer function, we can design a simpler model in Simulink and save computation resources once we proceed with the code implementation. With all these considered, the first order transfer function are the following:

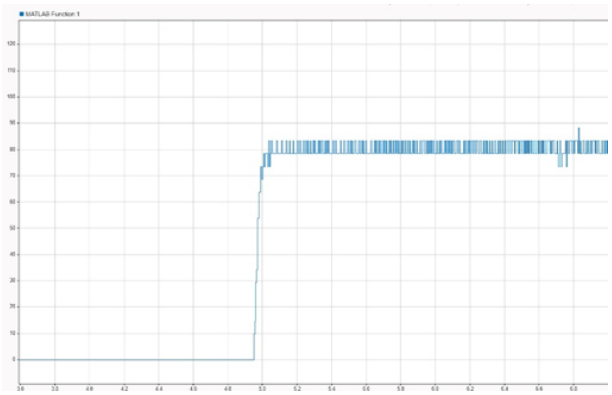
- front left: $G(s) = \frac{667.2}{s+44.93}$;
- front right: $G(s) = \frac{738.5}{s+52.75}$;
- rear left: $G(s) = \frac{709.8}{s+49.3}$.
- rear right: $G(s) = \frac{602.7}{s+44.98}$;

In **Figure 3** are reported the original acquisition from the encoder with Simulink: in the first row are reported the front left (**3a**) and right (**3b**) response, while in the second row are reported the rear left (**3c**) and right (**3d**) response.

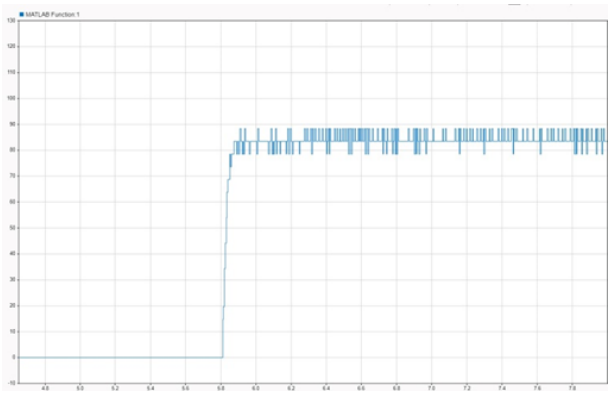
With the transfer function computed we can now calculate the maximum output that each motor can physically reach, when applied the maximum nominal voltage possible of 12V. To do this, we need to simply compute the $G(0)$ and multiply this for 12V. The results are reported below:

- front left: $G(0) \cdot \text{maxV} = \frac{667.2}{44.93} \cdot 12 \approx 178\text{RPM}$;
- front right: $G(0) \cdot \text{maxV} = \frac{738.5}{52.75} \cdot 12 \approx 168\text{RPM}$;
- rear left: $G(0) \cdot \text{maxV} = \frac{709.8}{49.3} \cdot 12 \approx 173\text{RPM}$;
- rear right: $G(0) \cdot \text{maxV} = \frac{602.7}{44.98} \cdot 12 \approx 161\text{RPM}$.

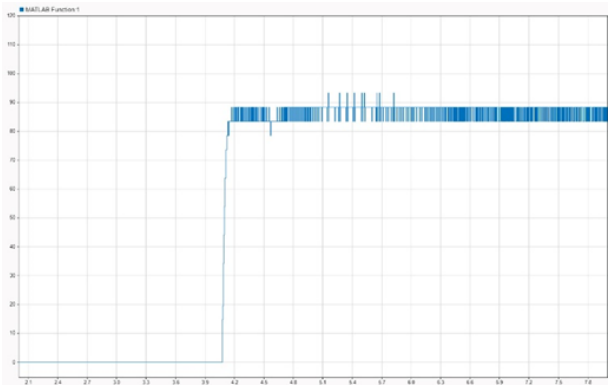
It's visible from these results that both the left motors are faster than the right one. These behaviour can be disruptive for a system that must have four motor that run at the same RPM. In addition, it's obviously not recommended to give more than 12V of voltage to the slower motor to speed them up and try to reach the fastest motor, since this behaviour could destroy those motor and compromise the system. For these reasons we choose to slow down the entire system to 160 RPM, which is a speed that every motor can reach.



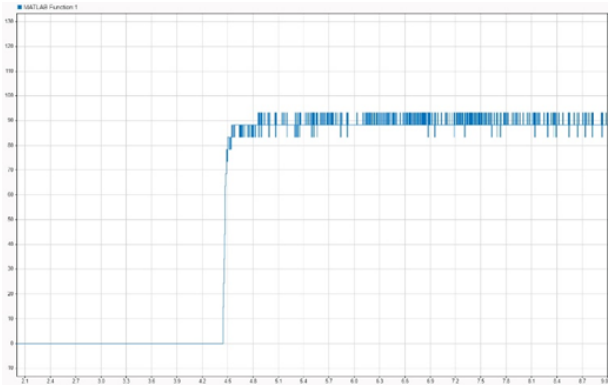
3a - Front left motor



3b - Front right motor



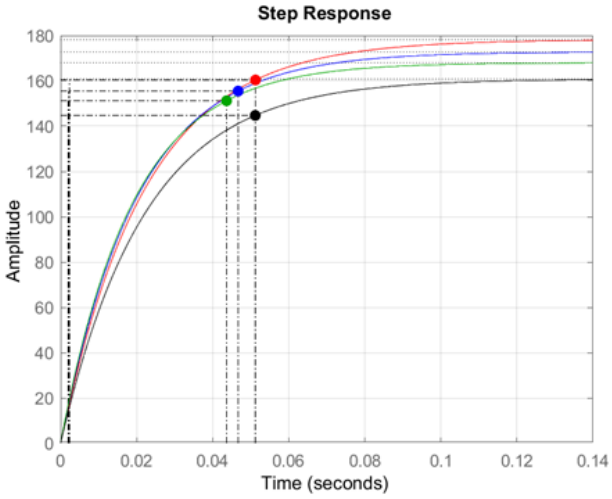
3c - Rear left motor



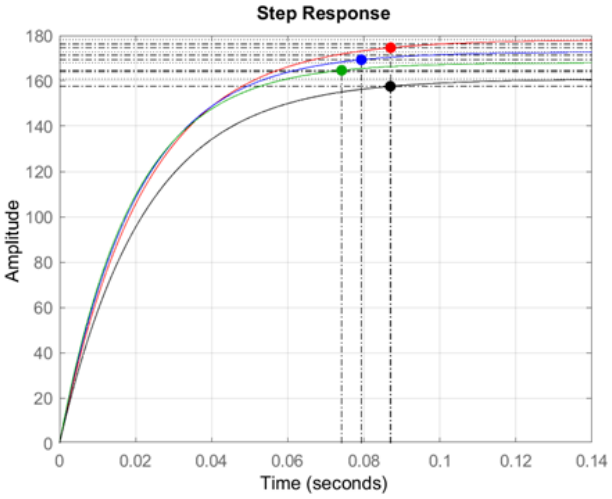
3d - Rear right motor

Figures 3: Step-response of the four motor

For a better results analysis, the Matlab plot of the step response of the same functions are reported in **Figures 4**, with **4a** underlining the rise time and **4b** showing the settling time, for each motor. In particular, in the plot can be seen in red the front left, in blue the rear left, in green the front right and in black the rear right.



4a - Motors TF - Rise time



4b - Motors TF - Settling time

Figures 4: Motors Matlab plots

The rise-time values have a range of [0.0416, 0.0489]s, while for the settling time the range span between [0.0742, 0.0871]s, all reasonable values for our DC motors. The slower motor, have a settling time of $0.0871s$, a value that will be important in the sampling time computing, calculate further in the project.

2.2 PI controller design

This section will cover the controller design, using the Matlab tool "Control System Designer". All controllers has been designed with the same parameters of response time and robustness.

A PID is a type of controller widely used in automatic control systems, where “PID” stands for Proportional, Integral and Derivative. The PID controller acts by calculating the error between the desired (setpoint) value and the actual (measured) value of a system, and applies three corrective actions based on three parameters Kp, Ki, Kd. Recall the general form of a digital PID:

$$\begin{aligned} s(k) &= s(k-1) + e(k) \\ u(k) &= Kp \cdot e(k) + Ki \cdot [s(k)] + Kd \cdot [(e(k) - e(k-1))]. \end{aligned} \tag{2.1}$$

The proportional term acts in proportion to the current error. If the error is large, the correction will be larger, if the error is small, the correction will be smaller; so the proportional gain Kp determines how strongly the controller reacts to the error.
The integral term takes into account the sum of past errors over time, trying to eliminate the stationary error (the residual error that might remain constant). The integral gain Ki determines how strongly the accumulated error over time affects the correction.
The derivative term acts on the change in error, that is, the rate at which the error changes. The derivative gain Kd anticipates the future behavior of the system and can reduce rapid oscillations or variations.

As already mentioned before, the controllers are designed as simple PI (Proportional-Integral), which are ideal for controlling transfer function of the first order, since the derivative part, in general, is useful in controlling oscillations around the stationary point that, as seen in the **Figures 4**, our functions do not have. The first-order system in fact is characterized by an exponential response to an input of a step type. It is a relatively simple system, with a response that has a rapid initial growth and tends toward an equilibrium value without oscillations. Our systems, in fact, reach the steady state point without overshooting or complex dynamics in a, relatively speaking, slow way. The scheme used to design the controllers is the classic feedback scheme, as seen in **Figure 5**.

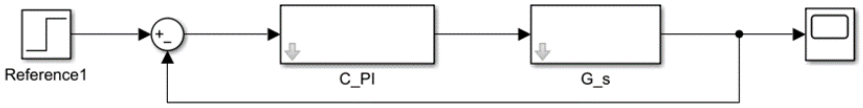


Figure 5: Feedback control scheme

The reference in this case is a step signal, that start from 0 and reach the value of 160 (the maximum RPM), while the last block is a simple scope to monitor the behaviour. This scheme will be used in the construction of the full system model in Simulink, later into the project.

For the project two types of controllers has been developed: fast controllers and slow controllers. This because, for our experiences with the system in the past courses and exams, sudden accelerations makes the system slips on the ground, especially on slippery ground like the sheath-like floor in our university.
The fast controllers have been designed with with a response time value of 0.05 and a robustness of 0.9, which is the maximum value, obtaining the following controllers:

- front left PI: $C(s) = 0.059952 \cdot \frac{(s+44.93)}{s}$;
- front right PI: $C(s) = 0.054164 \cdot \frac{(s+52.75)}{s}$;
- rear left PI: $C(s) = 0.056354 \cdot \frac{(s+49.3)}{s}$;
- rear right PI: $C(s) = 0.066368 \cdot \frac{(s+44.98)}{s}$.

The slow controllers have the same value of robustness as the fast one, but a response time value of 0.138, which led to obtaining the following controllers:

- front left PI: $C(s) = 0.021725 \cdot \frac{(s+44.93)}{s}$;
- front right PI: $C(s) = 0.019625 \cdot \frac{(s+52.75)}{s}$;
- rear left PI: $C(s) = 0.020418 \cdot \frac{(s+49.3)}{s}$;
- rear right PI: $C(s) = 0.024046 \cdot \frac{(s+44.98)}{s}$.

Lastly the bandwidth at $-3dB$ of the four loops has been computed, using the following equation for the closed-loop transfer function:

$$W(s) = \frac{C(s) \cdot G(s)}{1 + C(s) \cdot G(s)},$$

and plotting it in the Bode diagram to find the frequency at which the magnitude drop by $3dB$. Below are reported two bode diagram computed on the front left motor, are shown in **Figures 6** and **7**: the first one calculated with a fast PI controller, the second one with a slow PI controller.

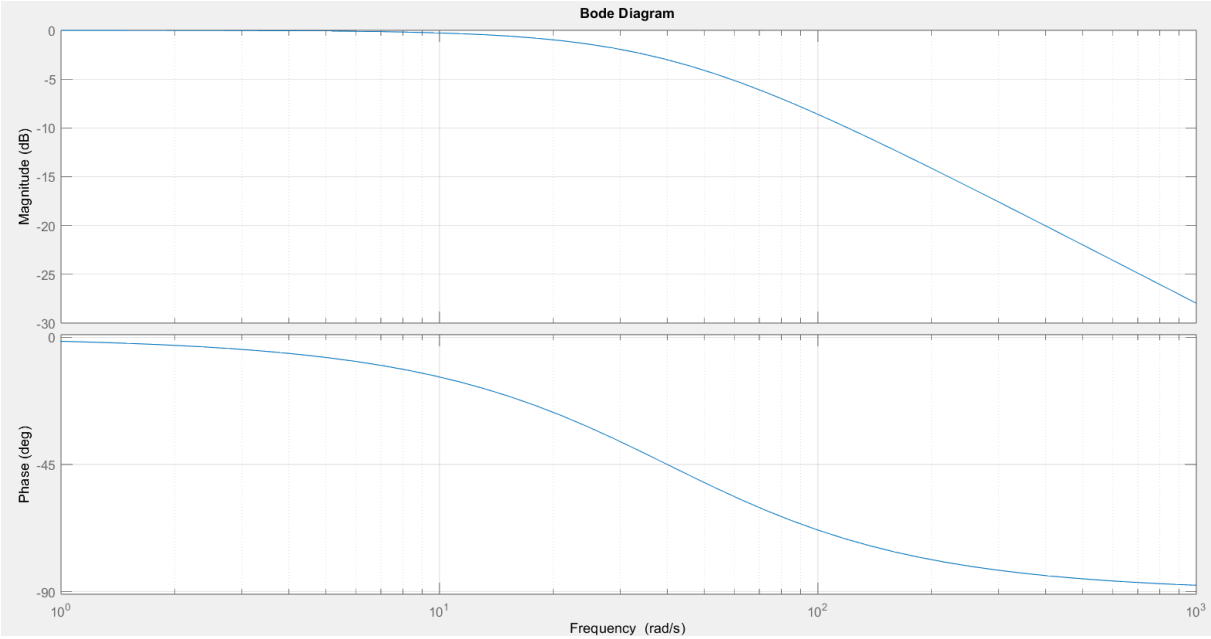


Figure 6: Bode for the front left fast controlled motor

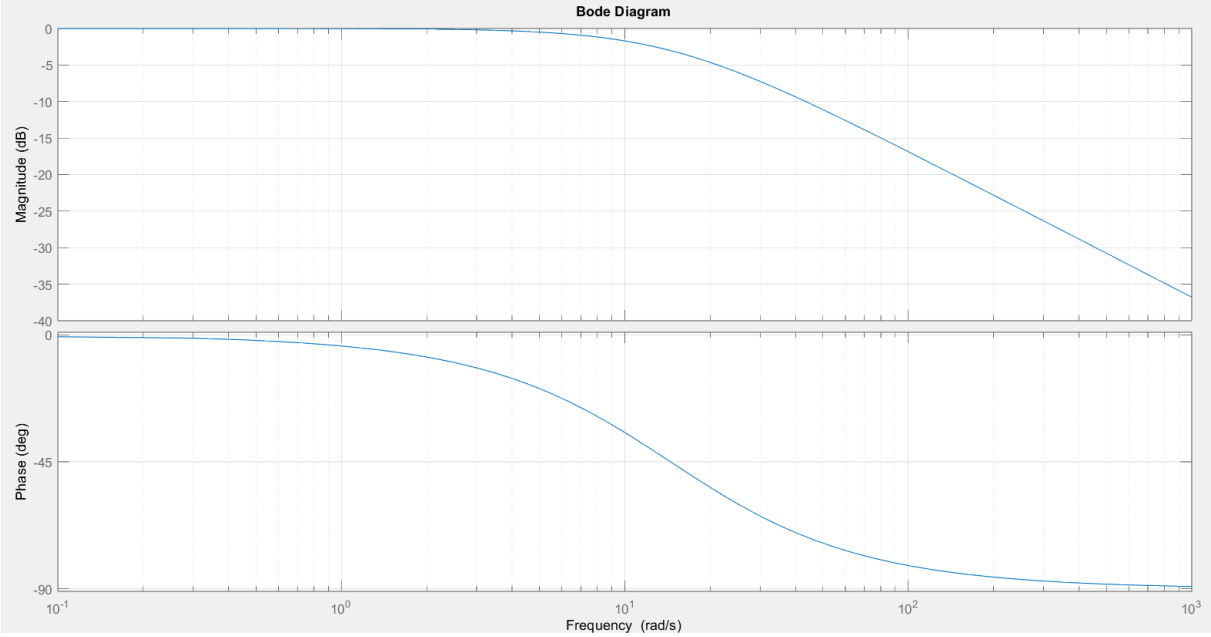


Figure 7: Bode for the front left slow controlled motor

This is important to verify that the chosen sampling time is adequate, which is discussed further in the report. The results, for all the controllers, are reported in the **Table 3** below:

Motor	**Fast C(s) ω_3 [rad/s]**	**Slow C(s) ω_3 [rad/s]**
RF	39.9	14.4
LF	39.9	14.4
RR	39.9	14.4
LR	39.8	14.5
FAx	39.9	14.4
RAx	39.8	14.5

Table 3: Bandwidth for all the motors control loop

The controllers validation is part of the MIL process, first step of the MBD. Now that we have functions and controllers for each motors, is indeed possible to develop and test a model in Simulink.

3. Model design and MIL validation

In this chapter will be reported the design of the model and it's validation through the MIL process. Both these activities have been carried out in Simulink.

3.1 Model development

The model developed for the rover system is composed of six PI controllers: one for each motor and two additional controllers for the front and rear axle control. These last two controllers were mandatory, in order to ensure that on the same axis the motor run at the exact same speed, measured in RPM. The input of the four main chains is the same step-like reference, simulating the input of the BT joypad. The input of the axle controller, instead, it's the difference between the output of the motors in that axle, while the axle controller's output is fed directly into PI controller output of the slave motor. The model is reported in **Figure 8**, with all the transfer function already loaded in the LTI blocks.

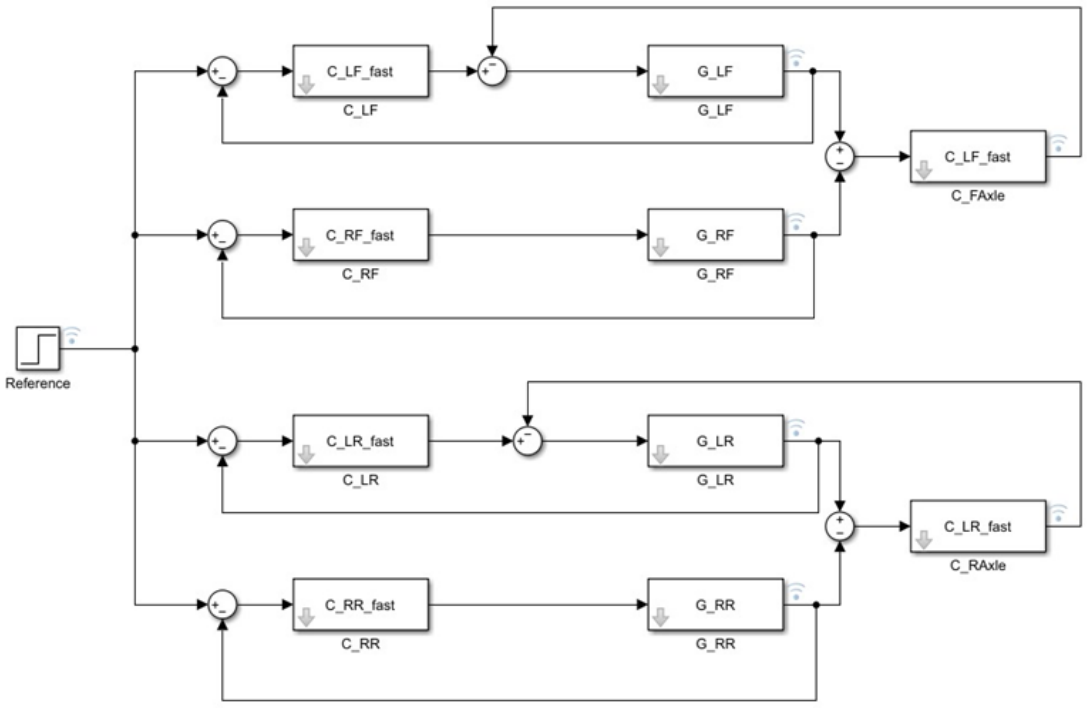


Figure 8: System model

As can be seen, the PI controllers for the axles are the same controllers as those controlling the left motors. This is because, as stated in the second chapter, the left motors are faster than the right counterparts, so we use the right motors as masters and the left motors act as slaves, slowing down to match the master's speed. Then, the axle output is fed and added to the slave controller's output, as aforementioned.

The validation of the model starts with a simple step response, to check the behaviour of the complete system, and then the same step response is used but with disturbs added to the chain, to simulate a difference in speed due to rough terrain (or other causes). The step reference is applied at $t = 1s$, while the disturb, with a value of 20, is applied at $t = 3s$ and removed at $t = 6s$, using a switch and a clock. The disturb scheme is visible in **Figure 9**.

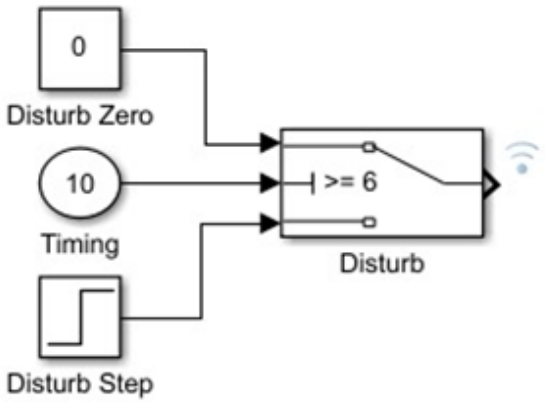


Figure 9: Disturb scheme

For completeness, in **Figure 10** is reported the complete model scheme, with the added disturb scheme.

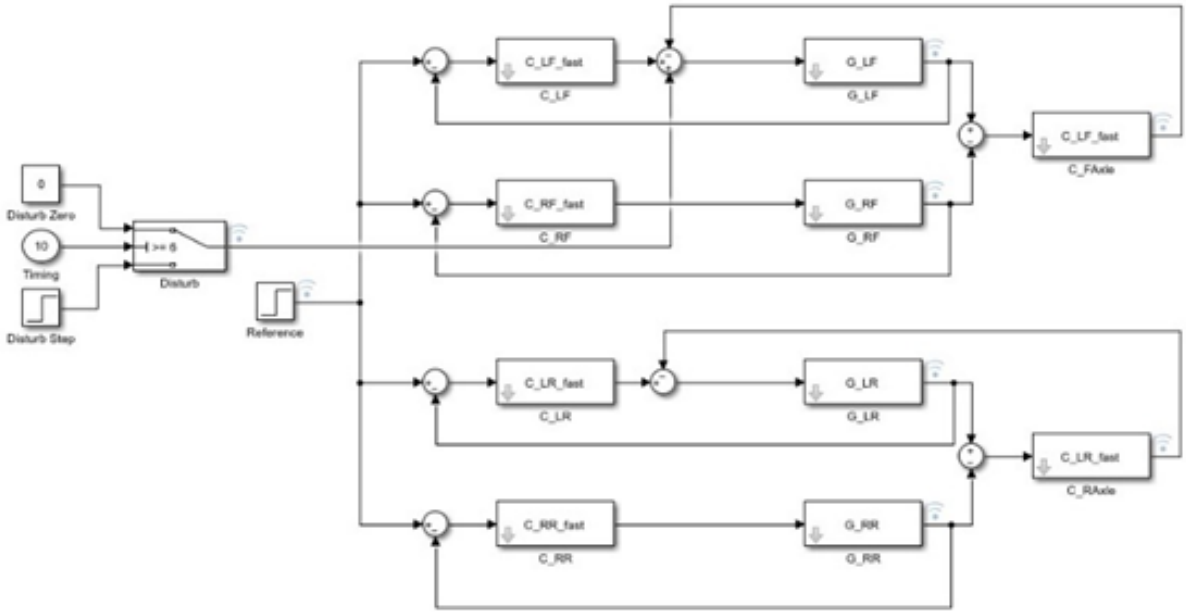
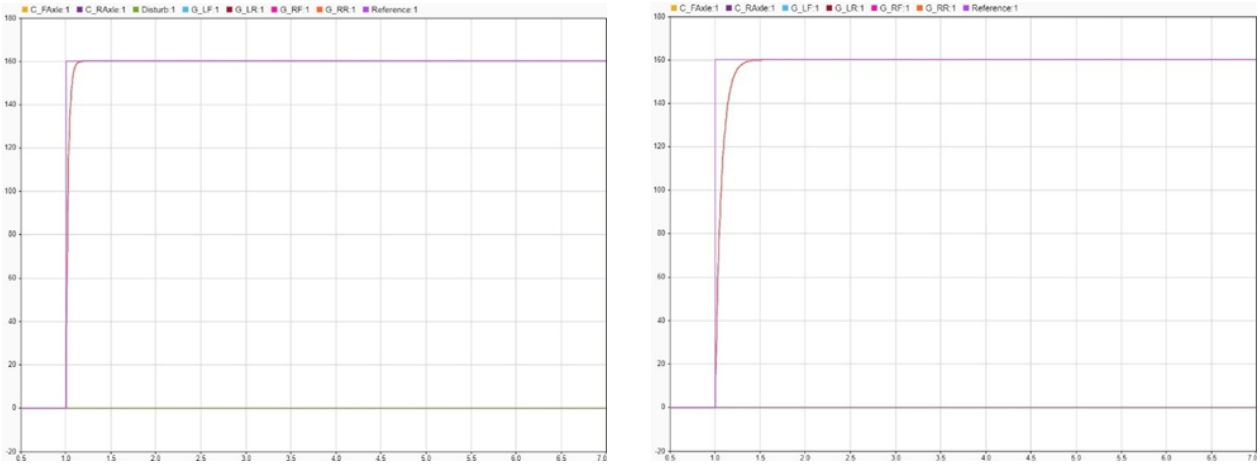


Figure 10: Model scheme with disturb

The disturb has been added to all the chain with the same method visible in above: summed to the controller output, before feeding the value into the process.

3.2 MIL Validation

The step disturbance introduces a sudden change in the motor control signal and is used to test the robustness of the control system and thus observe how this disturbance affects both the single motor and the axes behaviour, allowing us to observe whether the controllers compensate for this imbalance. Furthermore, by seeing the progress of the compensating action we can decide whether our controllers are fast enough and if they manage to return the system to a steady state. In **Figures 11**, we can see the output of the model, without the disturbance applied: in particular, in **11a** we can observe the fast controllers, while in **11b** we can see the slow controllers.

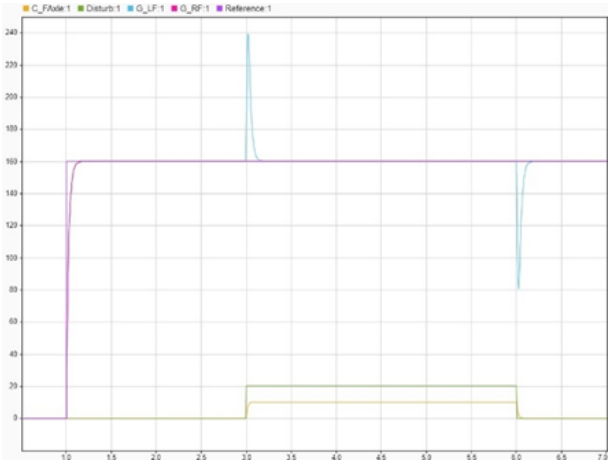


11a - Fast controlled system step-response

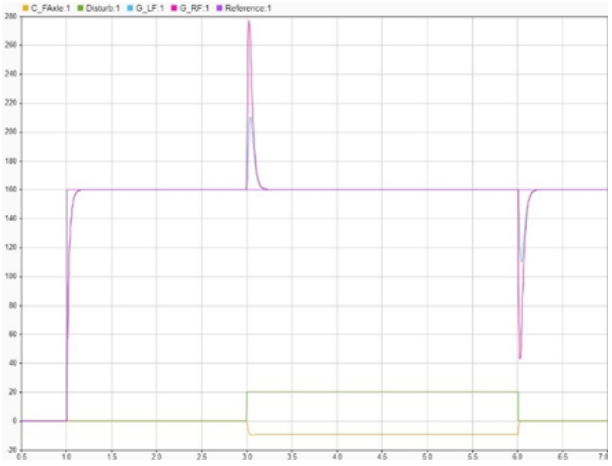
11b - Slow controlled system step-response

Figures 11: Step response of the whole system

We can already tell the differences between the two system by seeing these two plot: faster controllers were designed as fast as possible without introducing overshoots and slow controllers were designed as slow as possible without taking too much time to reach steady state, aiming to stay under 5 ms. Now that we know that the controllers responds well to a simple step stimulus, it's time to test the system robustness, introducing the disturbance. Plot outputs can be seen in **Figure 12**, reported in the next page: **12a** show the disturb applied to the front left motor, acting as a slave, and **12b** show the disturb applied to the front right motor, acting as the master of the front axle.



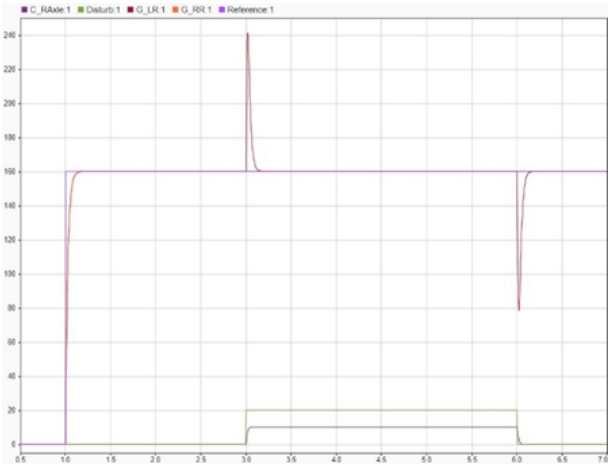
12a - Left front motor with disturbance



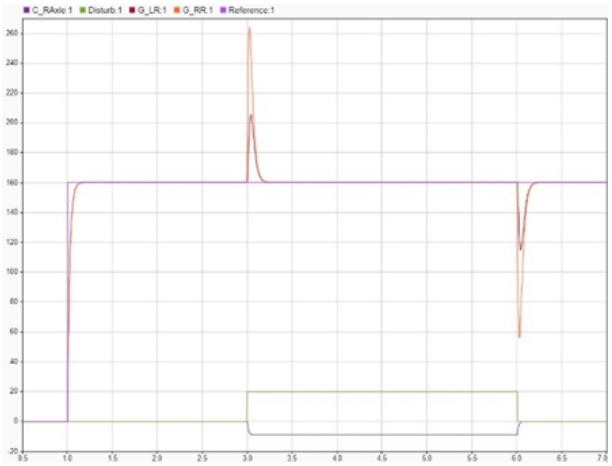
12b - Right front motor with disturbance

Figures 12: Disturbance applied on the front axle with fast PI controllers

It's visible that the disturbance applied to the master motor has a greater impact on both motors than the one applied only to the slave one. However the PI controllers managed to bring the axle back to the steady state, in all occasion. The same behaviour can be observed on the rear axle, with plots in **Figure 13a** and **13b**.



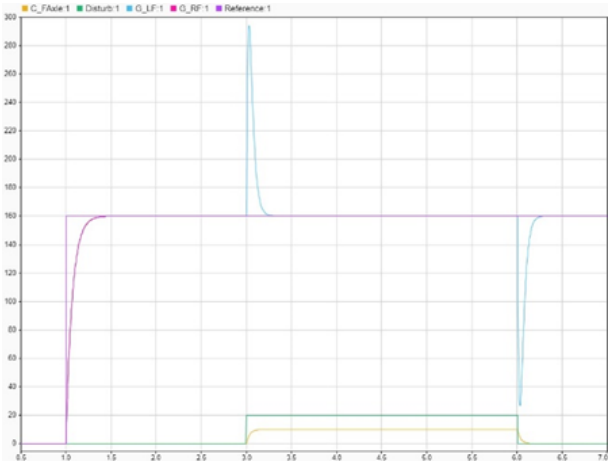
13a - Left rear motor with disturbance



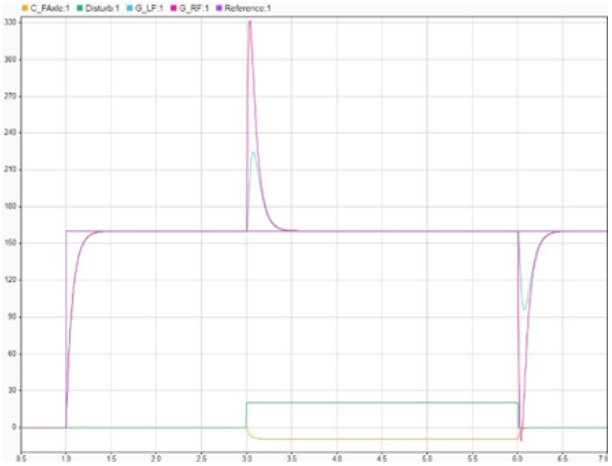
13b - Right rear motor with disturbance

Figures 13: Disturbance applied on the rear axle with fast PI controllers

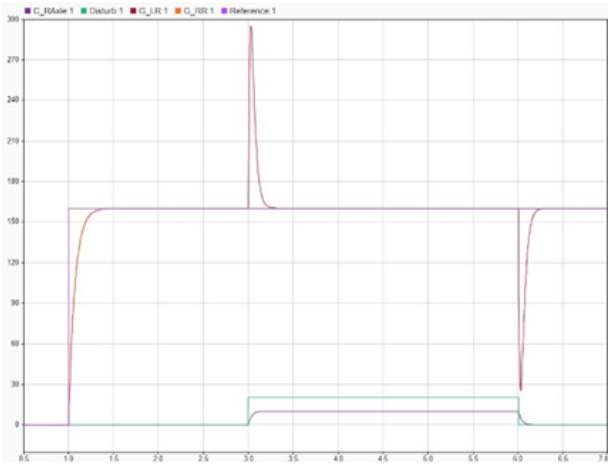
The recovery capability of the system with a fast PI controller is remarkable: within about 0.30 seconds both motors return to a steady state, demonstrating the readiness of the control system to absorb and correct the effect of the disturbance. In **Figures 14** are reported the effect of the disturbance on the system with slow PI controllers.



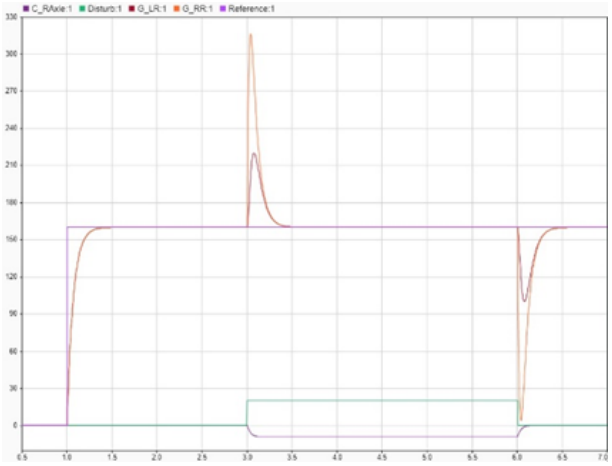
14a - Left front motor with disturbance



14b - Right front motor with disturbance



14c - Left rear motor with disturbance



14d - Right rear motor with disturbance

Figures 14: Disturbance applied on the system with slow PI controllers

The recovery time that the system with a slow PI controller takes for both motors to return to a steady state is approximately 0.70 seconds for both motors. By applying the disturbance on the left motor, the system is able to restore the steady state in an even faster time of approximately 0.37 seconds. The same behaviour is observed in the rear axle. These are obviously longer times than for the fast controllers, but still acceptable. It can be seen that disturbances applied to the system with slow controllers generate more significant overshoot at the time of the disturbance than that generated in the system controlled with fast controllers. This is probably because, as the controllers are slower, they take longer to kick in, leaving the motor at the mercy of the disturbance for a longer time. However, all the controllers react well to the disturbance and we can say that the system has passed the MIL validation. Now the project can enter it's second phase: SIL validation.

4. Model discretization and SIL validation

The SIL validation, as already stated, needs the model to be discretized. This process requires to calculate the sampling time and redo the model by adapting it to use discrete transfer functions. These two process will be described in the next section.

4.1 Discretization process

The equation to compute the sampling time is the following:

$$T_s \leq \frac{T_p}{10},$$

where T_s is the sampling time and T_p is the $plant - time$, or in other words the settling time of the slowest machine in the plant. In our case, we choose $T_p = 0,0871[s]$, because our slowest motor has a settling time exactly equal to that number. So, our sampling time will be:

$$T_s \leq \frac{0,0871}{10} = 0,00871,$$

or less than 8ms. We choose a sampling time of 5ms, staying consistent with the sampling time used in the motors characterization.

Now the transfer functions can be discretized. We use Matlab for the computation: using the $c2d()$ function, passing to it the right parameters (TF name, sampling time and discretization algorithm), using the Tustin method as third parameter, we get the following discrete transfer function:

- Front left: $G(z) = \frac{(1.5z+1.5)}{(z-0.798)}$;
- Front right: $G(z) = \frac{(1.631z+1.631)}{(z-0.767)}$;
- Rear left: $G(z) = \frac{(1.58z+1.58)}{(z-0.7805)}$;
- Rear right: $G(z) = \frac{(1.354z+1.354)}{(z-0.7978)}$.

The same method has been applied for computing the discrete transfer function for the fast controllers:

- Front left fast PI: $C(z) = \frac{(0.06669z-0.05322)}{(z-1)}$;
- Front right fast PI: $C(z) = \frac{(0.06131z-0.04702)}{(z-1)}$;
- Rear left fast PI: $C(z) = \frac{(0.0633z-0.04941)}{(z-1)}$;
- Rear right fast PI: $C(z) = \frac{(0.07383z-0.0589)}{(z-1)}$;

and for the slow controllers:

- Front left slow PI: $C(z) = \frac{(0.02417z - 0.01928)}{(z-1)}$;
- Front right slow PI: $C(z) = \frac{(0.02221z - 0.01704)}{(z-1)}$;
- Rear left slow PI: $C(z) = \frac{(0.02293z - 0.0179)}{(z-1)}$;
- Rear right slow PI: $C(z) = \frac{(0.02675z - 0.02134)}{(z-1)}$.

After these computations, we need to separate numerator and denominator for each function, in Matlab, to be proper used in the new Simulink design which exploits "Discrete Transfer Fcn" blocks instead of the classic "LTI" blocks, used in the MIL model. The new model is visible in **Figure 15**, below in the page.

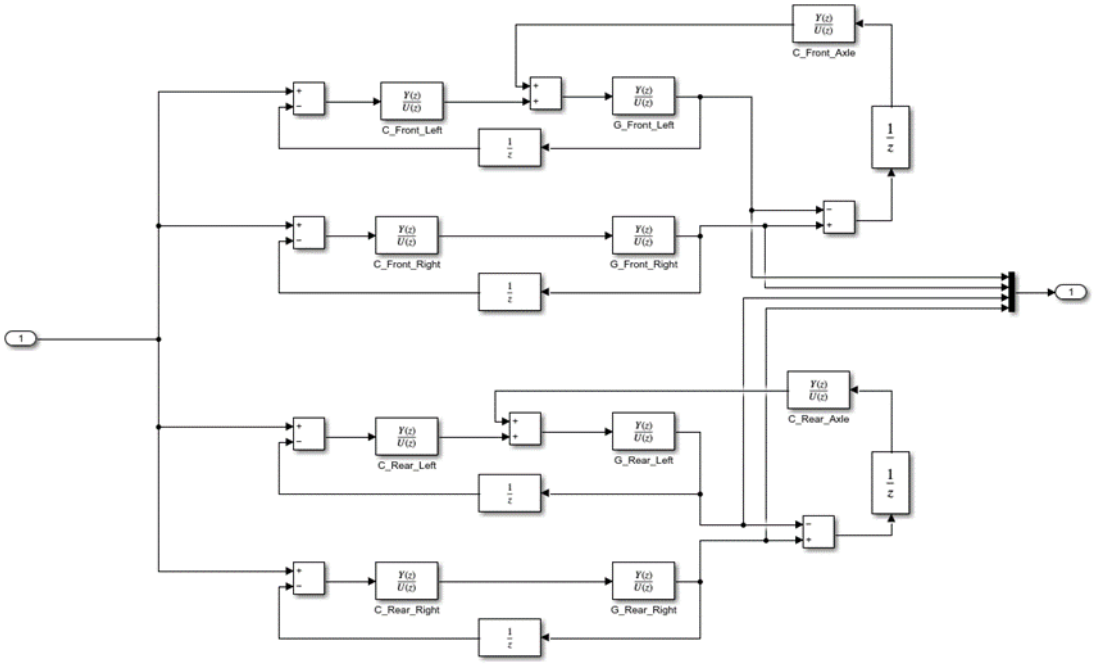


Figure 15: Discrete model

The new blocks need numerator, denominator and the sampling time as parameters.

4.2 SIL validation

After developing the new model, we can use it in the SIL/PIL scheme, visible in **Figure 16**.

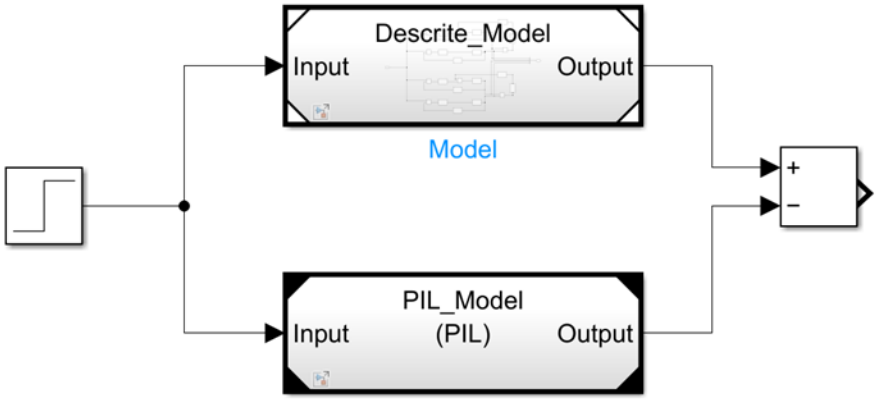


Figure 16: SIL/PIL scheme

This scheme implements the discrete model twice:

- Simulation Mode: simulates system behaviour in discrete time, but without running generated C code; it is an ideal simulation, similar to MIL;
- SIL/PIL Mode: this mode executes the generated C code (derived from the model) and tests how the code behaves when executed in the simulation ambient; it simulates what would happen when the code is executed on real hardware.

The objective of this set-up is to compare the ideal behaviour of the simulated controllers with the real behaviour of the generated code, the difference between the two outputs is then calculated, to check whether discrepancies exist. If the difference is zero or negligible, it can be concluded that the generated code is accurate and faithfully reproduces the behaviour predicted by the simulation. This is a crucial step to ensure that the discretization and controller implementation does not introduce significant errors or delays. The same step input as in previous simulations is used, starting at 0 and rising to 160 with a start time set at 3 seconds.

The results of the SIL tests are reported in the next page. In particular, **Figure 17** report the results for the SIL test executed on the model with fast PI controllers, while **Figure 18** shows the result for the SIL test executed on the model with slow PI controllers. Both the models pass the tests: there is no difference between the simulation and the C code behaviour, as shown in both figures.

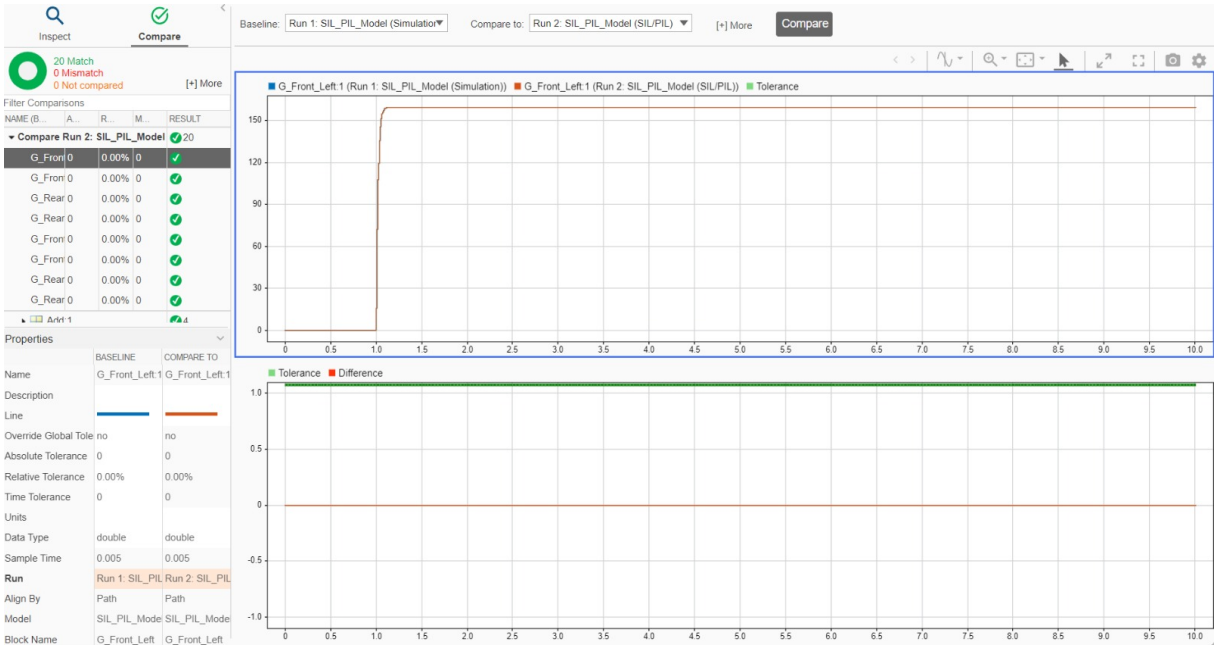


Figure 17: SIL validation for fast controllers

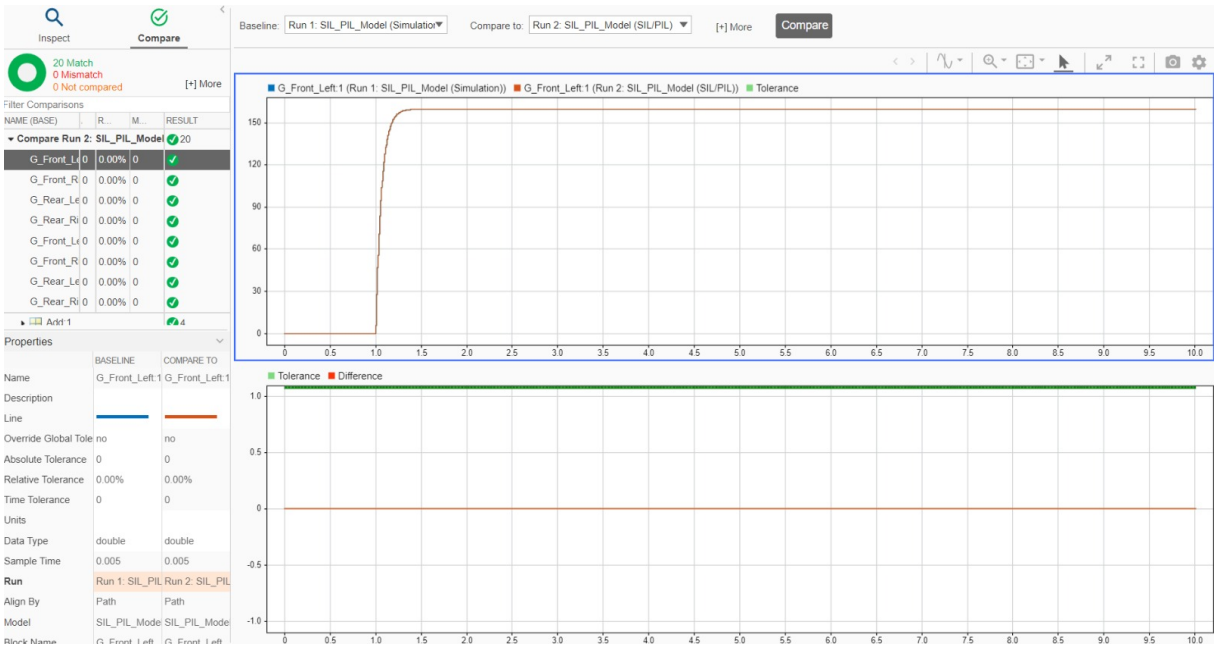


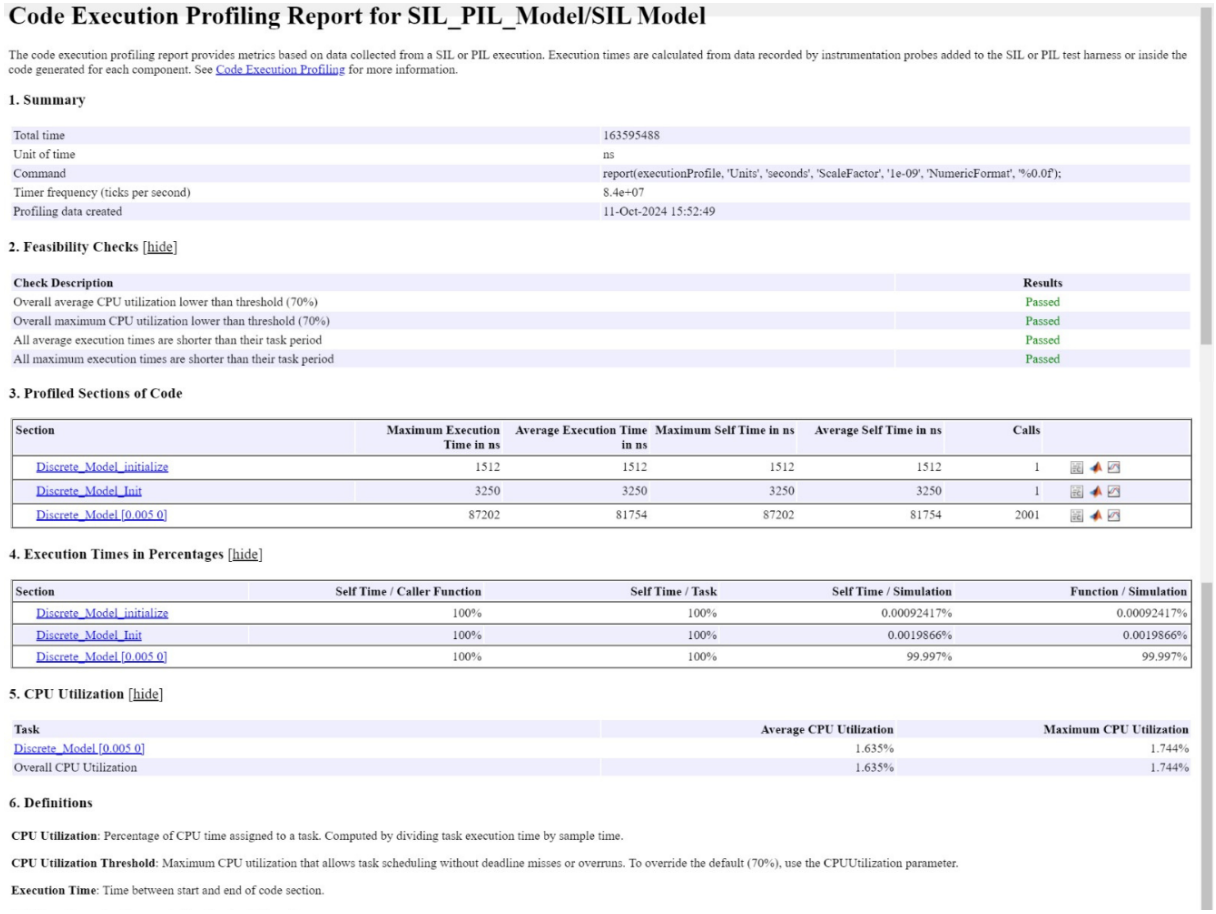
Figure 18: SIL validation for slow controllers

As can be seen, the differences between the simulation and the C code SIL test is 0% for all the motors. The SIL validation is passed and now the project can proceed into it's third step: PIL validation.

5. PIL validation

PIL validation is an important part of the process: it test if the chosen processor is able to run the code, in out case the "STM32F401RE". Unlike the SIL phase, where the code is simulated in a test environment on the computer, PIL validation involves the actual execution of the code on physical hardware. In this part, the generated code will be deployed on the board and an automated validation will check various aspects of the running code. In addition, the test checks whether the generated code runs correctly on the hardware, without compilation or runtime errors, and measures processor utilisation and execution time to verify that the code is efficient enough to be executed without overloading the system.

The process make uses of the same model in **Figure 16**, but this time, the bottom part will be used in PIL mode. This allows the simulation to test the board once the code is deployed on it, and check all the parameters. The report for the fast controlled system is in **Figure 19**.



Code Execution Profiling Report for SIL_PIL_Model/SIL Model

The code execution profiling report provides metrics based on data collected from a SIL or PIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL or PIL test harness or inside the code generated for each component. See Code Execution Profiling for more information.

1. Summary

Total time	163595488
Unit of time	ns
Command	report(executionProfile, 'Units', 'seconds', 'ScaleFactor', '1e-09', 'NumericFormat', '%0.0f');
Timer frequency (ticks per second)	8.4e+07
Profiling data created	11-Oct-2024 15:52:49

2. Feasibility Checks [hide]

Check Description	Results
Overall average CPU utilization lower than threshold (70%)	Passed
Overall maximum CPU utilization lower than threshold (70%)	Passed
All average execution times are shorter than their task period	Passed
All maximum execution times are shorter than their task period	Passed

3. Profiled Sections of Code

Section	Maximum Execution Time in ns	Average Execution Time in ns	Maximum Self Time in ns	Average Self Time in ns	Calls
Discrete_Model_initialize	1512	1512	1512	1512	1
Discrete_Model_Init	3250	3250	3250	3250	1
Discrete_Model [0.005 0]	87202	81754	87202	81754	2001

4. Execution Times in Percentages [hide]

Section	Self Time / Caller Function	Self Time / Task	Self Time / Simulation	Function / Simulation
Discrete_Model_initialize	100%	100%	0.00092417%	0.00092417%
Discrete_Model_Init	100%	100%	0.0019866%	0.0019866%
Discrete_Model [0.005 0]	100%	100%	99.997%	99.997%

5. CPU Utilization [hide]

Task	Average CPU Utilization	Maximum CPU Utilization
Discrete_Model [0.005 0]	1.635%	1.744%
Overall CPU Utilization	1.635%	1.744%

6. Definitions

CPU Utilization: Percentage of CPU time assigned to a task. Computed by dividing task execution time by sample time.

CPU Utilization Threshold: Maximum CPU utilization that allows task scheduling without deadline misses or overruns. To override the default (70%), use the CPUUtilization parameter.

Execution Time: Time between start and end of code section.

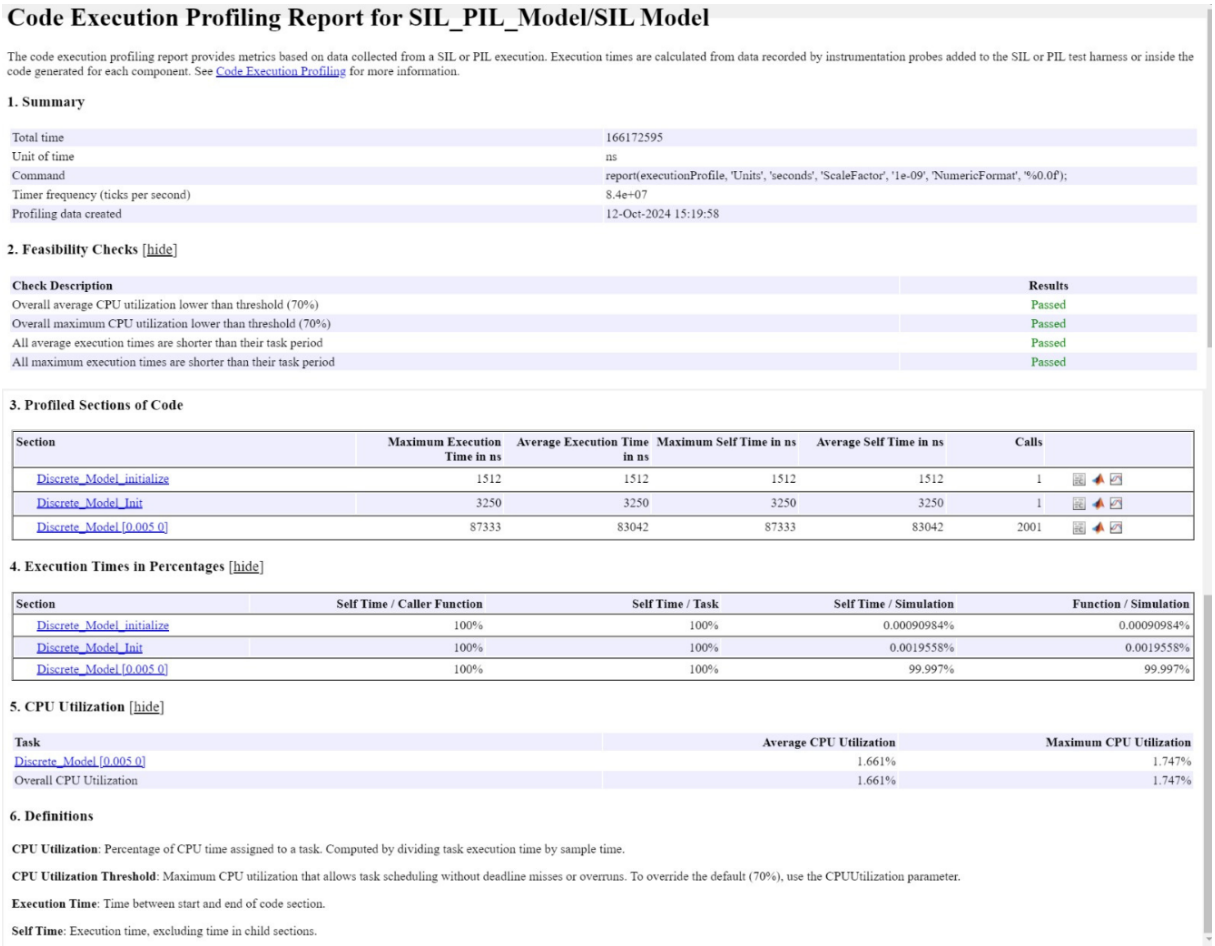
Figure 19: PIL validation for fast controllers

The report shows that the fast controlled system pass all the tests:

- both the overall and maximum CPU utilization is below the 70% threshold; this means that, ideally, the design could be implemented using a Real Time Operating System (RTOS), such as FreeRTOS (supported by our microcontroller) with tasks;
- all the execution time, both average and maximum, of the tasks (such as initialization functions and the design run itself) are shorter than their task period, meaning that there are no deadline misses (another piece of evidence supporting the possible use of an RTOS);
- the overall CPU utilization is, in percentage, equal to 1.6%, with a maximum peak of 1.7%.

So, the fast controlled model passed all the PIL tests.

Concerning the slow controlled system, the PIl result can be seen in **Figure 20**.



Code Execution Profiling Report for SIL_PIL_Model/SIL Model

The code execution profiling report provides metrics based on data collected from a SIL or PIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL or PIL test harness or inside the code generated for each component. See Code Execution Profiling for more information.

1. Summary

Total time	166172595
Unit of time	ns
Command	report(executionProfile, 'Units', 'seconds', 'ScaleFactor', '1e-09', 'NumericFormat', '%0.0f');
Timer frequency (ticks per second)	8.4e+07
Profiling data created	12-Oct-2024 15:19:58

2. Feasibility Checks [hide]

Check Description	Results
Overall average CPU utilization lower than threshold (70%)	Passed
Overall maximum CPU utilization lower than threshold (70%)	Passed
All average execution times are shorter than their task period	Passed
All maximum execution times are shorter than their task period	Passed

3. Profiled Sections of Code

Section	Maximum Execution Time in ns	Average Execution Time in ns	Maximum Self Time in ns	Average Self Time in ns	Calls
Discrete_Model_initialize	1512	1512	1512	1512	1
Discrete_Model_Init	3250	3250	3250	3250	1
Discrete_Model [0.005 0]	87333	83042	87333	83042	2001

4. Execution Times in Percentages [hide]

Section	Self Time / Caller Function	Self Time / Task	Self Time / Simulation	Function / Simulation
Discrete_Model_initialize	100%	100%	0.00090984%	0.00090984%
Discrete_Model_Init	100%	100%	0.0019558%	0.0019558%
Discrete_Model [0.005 0]	100%	100%	99.997%	99.997%

5. CPU Utilization [hide]

Task	Average CPU Utilization	Maximum CPU Utilization
Discrete_Model [0.005 0]	1.661%	1.747%
Overall CPU Utilization	1.661%	1.747%

6. Definitions

CPU Utilization: Percentage of CPU time assigned to a task. Computed by dividing task execution time by sample time.

CPU Utilization Threshold: Maximum CPU utilization that allows task scheduling without deadline misses or overruns. To override the default (70%), use the CPUUtilization parameter.

Execution Time: Time between start and end of code section.

Self Time: Execution time, excluding time in child sections.

Figure 20: PIL validation for slow controllers

The results are nearly identical as those for the PIL test of the fast controllers: the only differences can be seen in the computation total time, being slightly larger in this second run, being 2577107[ns] slower, in the average CPU utilization, being 0.026% larger, and in the maximum CPU utilization, being 0.03% larger. Overall the two tests are basically the same, meaning that the PIL test for the slow controlled system has passed.

5.1 Consideration on SIL and PIL validations

Despite SIL and PIL being both crucial part in the validation of a model, the results obtained are unfortunately useless: they refers only to the model we provided, that takes in account only the four electric motors, but do not include the code, and the subsequently computation, needed for the PSX joypad. However, since the joypad uses an SPI communication (know for being a fast communication protocol) and taking in account the SIL and PIL validation results, we could say that adding the controller to the model would not jeopardize both the validations.
Simulations were carried out purely for completeness of design, since the course focused on the use of the MBD and was known that the HIL validation would not be carried out due to the fact that this would require a high-performance computer that would be wasted on such a simple model.

6. Switch model design

During the project, while testing the two types of controllers on the real system, it was noted that the fast controllers made the rover slip, which rotated a few degrees to the left. This behaviour was mitigated with the slow controllers, so a new model was designed that could take into account the switch from fast to slow controllers at runtime. The design is visible in **Figure 21**.

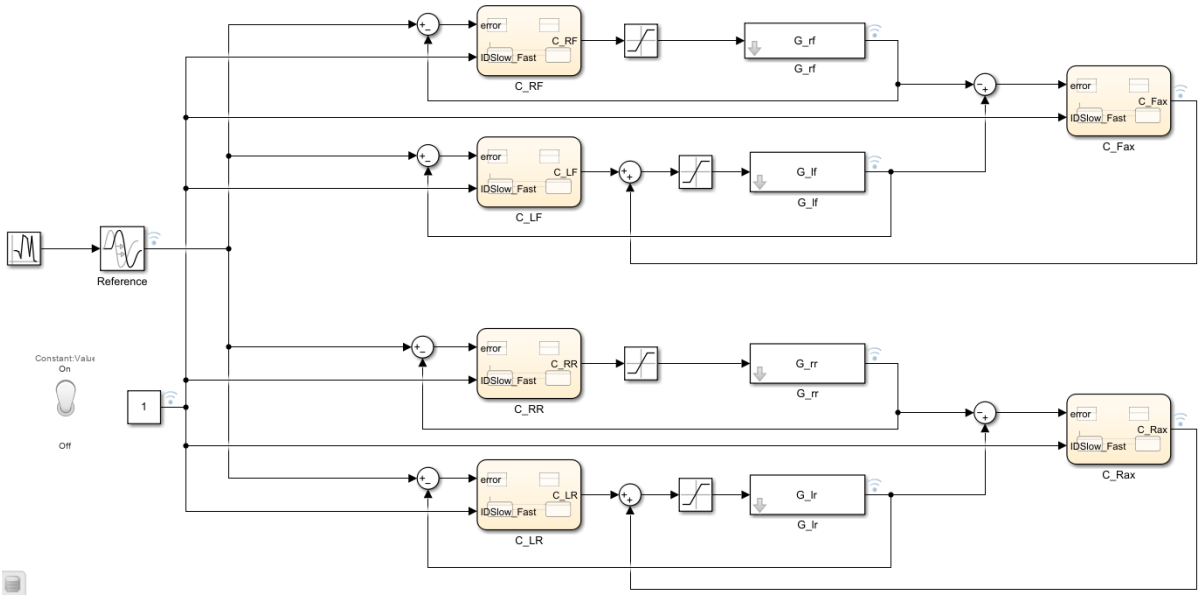


Figure 21: Switch control model

The new design implements the controller blocks as a Stateflow diagram. In addition, saturation blocks have been added to the controllers output line, limiting the outputs in the range accepted by the motors, which we recall being $[-12, 12]$. These diagrams have inside both the controllers, and the user can choose between them using the ID: 0 for fast and 1 for slow. In **Figure 22** is reported the Stateflow chart.

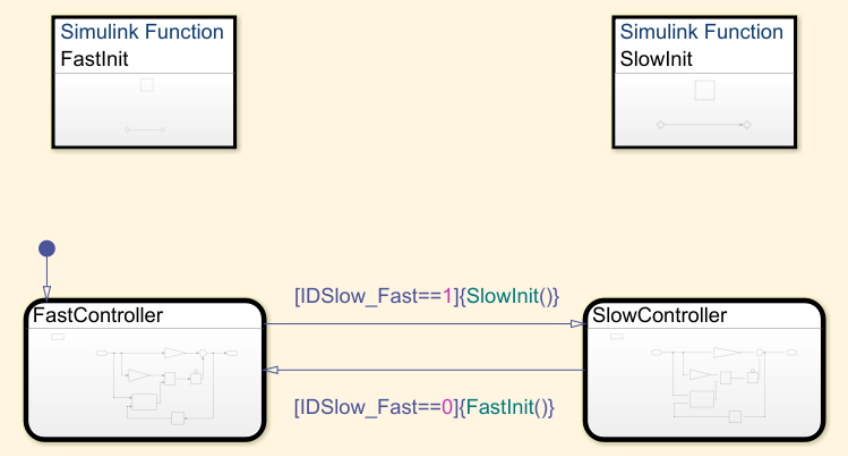


Figure 22: Inside the Stateflow chart

So each new controller takes in input two arguments: the error, computed as before, and the controller ID, connected to a switch in the model. For this model only a MIL validation as been carried out for various reason:

- the SIL and PIL for both the slow and fast controlled model has already been validated;
- introducing this switch, should be computationally lightweight, since in terms of C-code, it means that we needs only to switch the K_p and K_i parameters, assigning the new values at runtime;
- again the SIL and PIL validations will be not consistent since the joypad comunication is not taken in account.

6.1 MIL validation of the improved model

The MIL validation was carried out through several steps: the first one was a simple MIL validation, as seen in third chapter. The scheme used in the state to simulate the controller behaviour is visible in **Figure 23**.

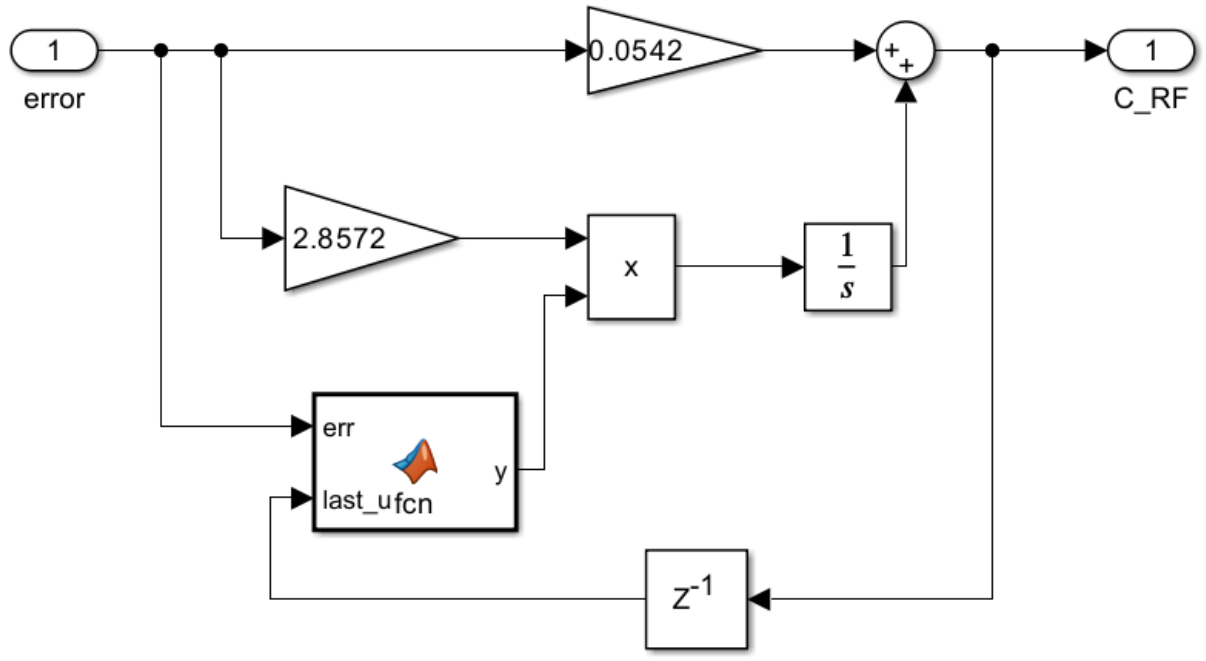


Figure 23: New PI controller scheme

The proportional and integrative coefficients can be clearly seen. In addition, a Matlab function that acts as the "anti windup" is added: the windup occurs when the PID controller integrator accumulates a high value due to a persistent error over time. This problem is common when the actuator reaches the limit of its physical capacity, and thus cannot provide the required response. To prevents this effect, we implemented the stop summation method which suspends the action of the integrator temporarily when the output is saturated, preventing excessive values from accumulating. In **Figure 24** is reported the Matlab code inside the function.

```
function y = fcn(err,last_u)

    if((last_u < 12 && err > 0)||(last_u > -12 && err < 0))
        y=1;
    else
        y=0;
    end
end
```

Figure 24: Anti wind-up method

The behaviour is simple to understand:

- if the error is positive and the last output was lesser than 12 (the maximum voltage possible) or if the error is negative and the last output was greater than -12, the the output of this function is 1, "enabling" the sum of the new integral term;
- in all the other cases, the output of this function is 0 that multiplyed with the integral term, nullify it and thus adding 0 to the output to the chain output.

This model, despite implementing controllers already validated in other schemes, however wasn't perfect: at the moment of the switch some bumps were recorded, as shown in **Figure 25**.

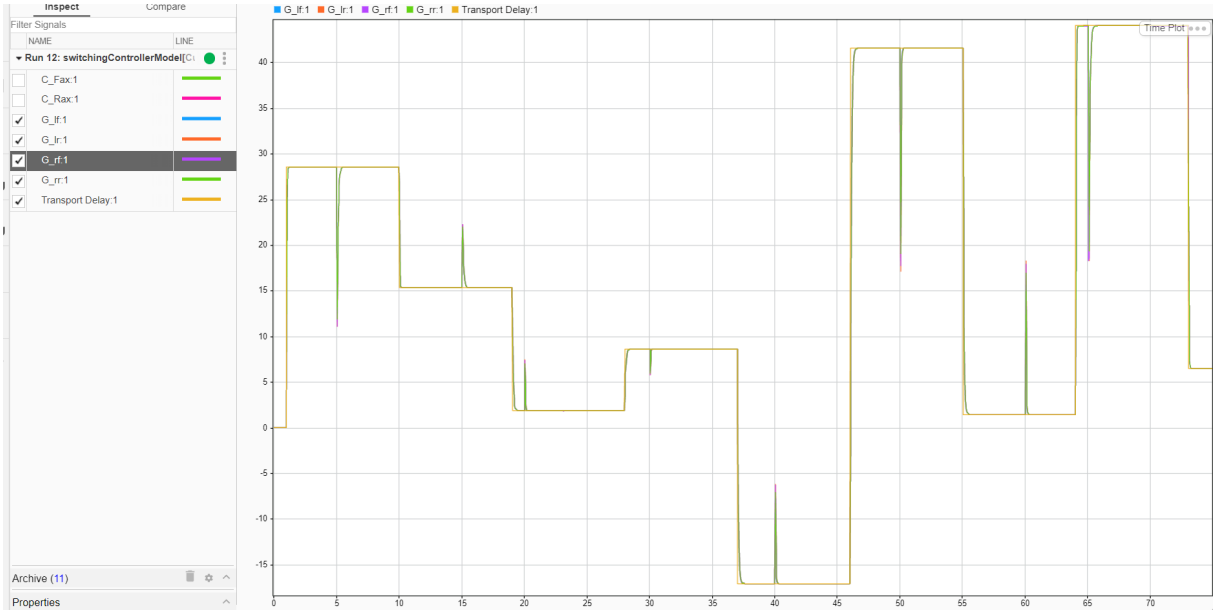


Figure 25: Simulation with anti wind-up, not bumpless

This effects are caused by the reset of the error value at the moment of the controller type change. This is a common behaviour in scheme that implements switch from manual to automated control and vice versa.

When changing controllers (from fast to slow or vice versa) without saving the status, the internal controller variables (such as the error integral in the case of a PI) may not be updated correctly. The integral term of the PI controller is responsible for the accumulation of the error over time; if it is not saved, the new controller may start with an incorrect integral value compared to the current system dynamics, causing an abrupt transient.
However, sorting out this type of behaviour is simple: if we save the state of the error, and then pass it to the new controller, the latter have an error value that starts from the right value again. To do this in Simulink, we simply add a Matlab function that save the state, as shown in **Figure 26**.

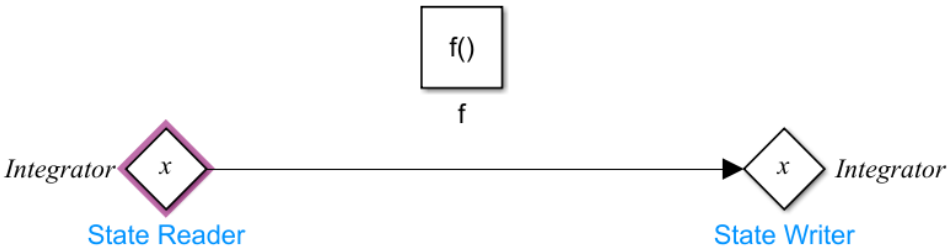


Figure 26: State saving function

The new controller become as shown in **Figure 27**.

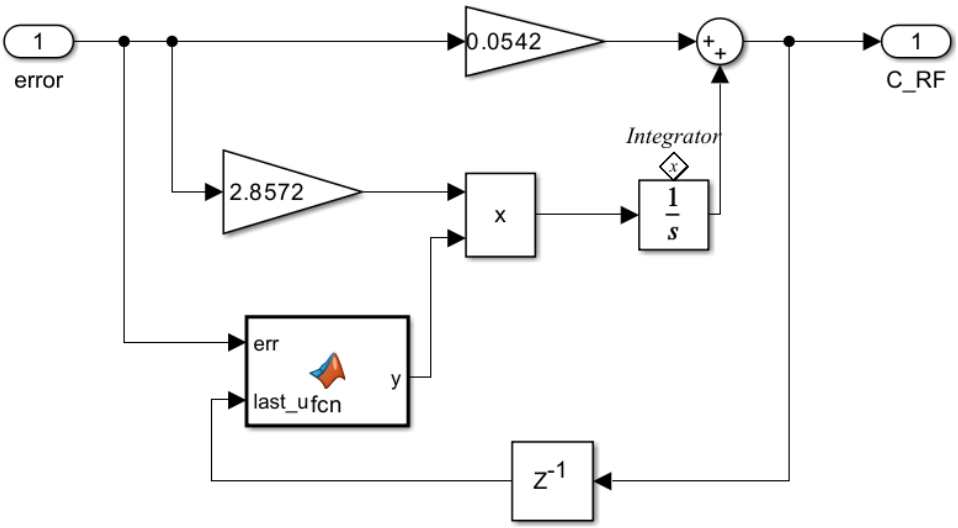


Figure 27: Improved controller scheme with bumpless behaviour

The new MIL test can be seen in **Figure 28**: as shown the bump that was present before have now disappeared, meaning that the controller is indeed bumpless. So the system is now ready: the motors reach steady state without bump and follow the reference perfectly.

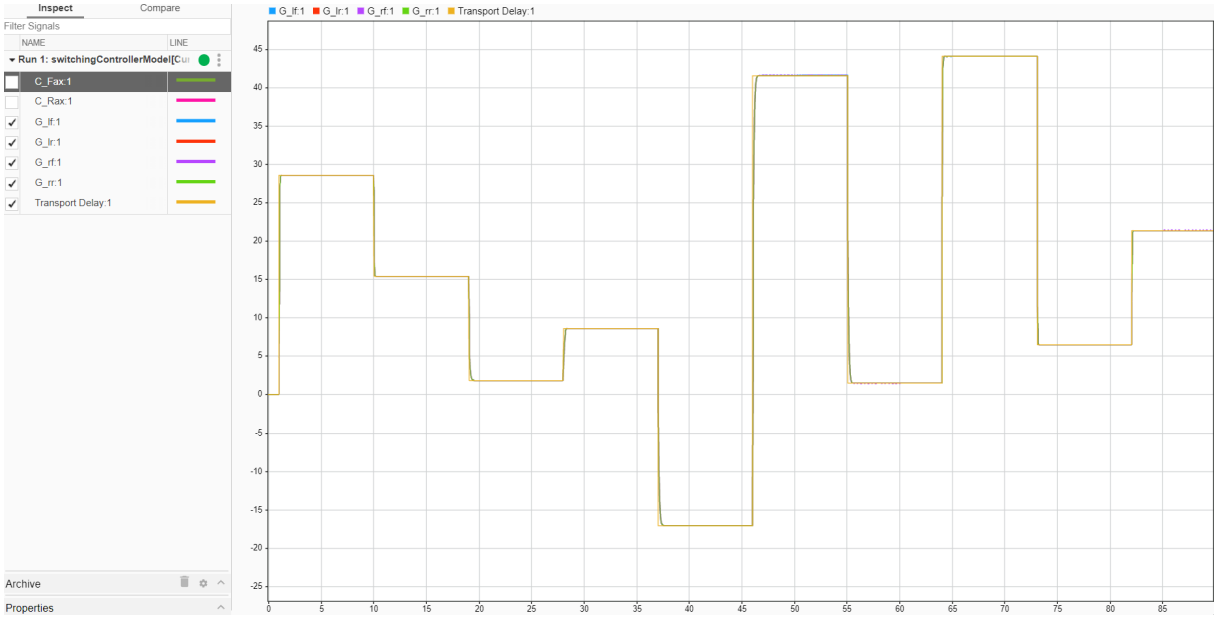


Figure 28: Simulation with bumpless behaviour

This is however a strange behaviour of the model, since we expected to find still some bumps on the simulation. In fact if the state is the same, being the fast K_i different from the slow K_i, the output will be different. This should in theory create some bumps on the switch, that the graph do not shows. We indeed found these bumps during the direct coding implementation part. To take down these effect we had to rescale the state passed to the new controller, with a factor computed as:

$$\frac{\text{old } K_i}{\text{new } K_i}.$$

7. Hardware description

The hardware used to set up the rover system is composed as follow:

- four electric DC motors model "36GP540-51";
- two motor drivers "Sabertooth 2x12";
- one microcontroller "STM32F401RE";
- one bluetooth controller "PSX Lynxmotion";
- one LiPo 12V battery.

The kit included four wheels, a chassis and various components used to build the whole system. The electric motors have already been characterised in chapter 2.
The controller is a Lynxmotion PS2/PSX with it's BT receiver. It uses SPI as communication protocol and it's code will be explained in the next chapter.
The STM32F401RE is a small but powerfull microcontroller, based on the high-performance ARM©Cortex© -M4 32-bit RISC core operating at a frequency of up to 84 MHz. It offers 10 timer interfaces, a large set of GPIO pins, several communication interfaces (UART/USART, SPI and I2C) and even RTOS compatibility (with FreeRTOS support and even external scheduler).
The "Sabertooth 2x12" are both versatile and efficient dual motor drivers. it's suitable for medium powered robots, controlling two motors each. They can be used in several ways including packetized and PWM modes. In particular, the PWM mode has been used in this project. The datasheet states that this mode needs the input signal to be filtered before reaching the motor driver itself. For this reason, following the advice on the datasheet, an RC filter has been dimensioned such as:

$$f_c = \frac{1}{2\pi RC} = \frac{1}{2\pi \times 10^4 \times 0,1 \times 10^{-6}} \approx 159[Hz,]$$

with:

- $R = 10 \times 10^3\,\Omega$;
- $C = 0,1 \times 10^{-6}\,F$;

with such filter, the high frequency components (the rapid changes between the ON and OFF states) are removed, allowing only the low-frequency components (the Duty Cycle average value) to pass through.

7.1 Motor driver setup

To use the PWM, the motor driver needs to be configured in "Analog mode". On the bottom part, a series of six switches allow to configure various aspect of the device:

- switches 1 and 2 need to be in the UP/ON position to enable the analog mode;
- the position of switch 3 depends on the type of battery used (but no differences were found between the two possible positions);
- switch 4 allow to choose between two mode of controlling the motors:
  - the UP position enable "Mixed mode", for easy steering of differential-drive vehicles; the signal fed into the S1 channel control the forward/back motion of the vehicle, and the analog signal fed into S2 controls the turning motion of the vehicle;
  - the DOWN position enable "Indipendend mode", in which the signal fed to S1 directly controls Motor 1 and the signal fed to S2 controls Motor 2; since our system has no differential, this mode has been chosen;
- switch 5 allows to choose between an exponential response to the input, if put in the DOWN position, or a linear response, if put in the UP position; we choose the linear response, so the switch is in the UP position;
- switch 6, if put in the DOWN position, enables the "4x sensitivity mode", where the input signal range between $[1.875, 3.125]V$ range, with a zero point of $2.5V$; in the UP position the input spans the range $[0, 5]V$, with again a zero point of $2.5V$; for our purpose, the switch is set in the DOWN position.

7.2 STM32 project setup

The first step in implementing the algorithm is to create the project in CubeIDE. This means that we have to choose all the timers and other project features to run on our STM32F401. In **Figure 29** is shown the IOC (Input/Ouput Configuration) of the project, with all the necessary timers and peripherals activated.
The chosen timer for the system, accountable for managing the callback function, is the TIM9, while the other timers, with their respective channels, are assigned to the encoder and to the four PWM signals. The PWM needs only one channel per timer, while the encoders needs two channel for each timer (set in encoder mode in the IOC). The PWM timer list is as follows:

- TIM2 Channel 1: PWM signal for the front right motor;
- TIM2 Channel 2: PWM signal for the front left motor;
- TIM11 Channel 1: PWM signal for the rear right motor;
- TIM10 Channel 1: PWM signal for the rear left motor;

while the encoder list is as follows:

- TIM1, uses channels 1 and 2 for the front right motor encoder;
- TIM3, uses channels 1 and 2 for the rear right motor encoder;
- TIM4, uses channels 1 and 2 for the front left motor encoder;
- TIM5, uses channels 1 and 2 for the rear left motor encoder;

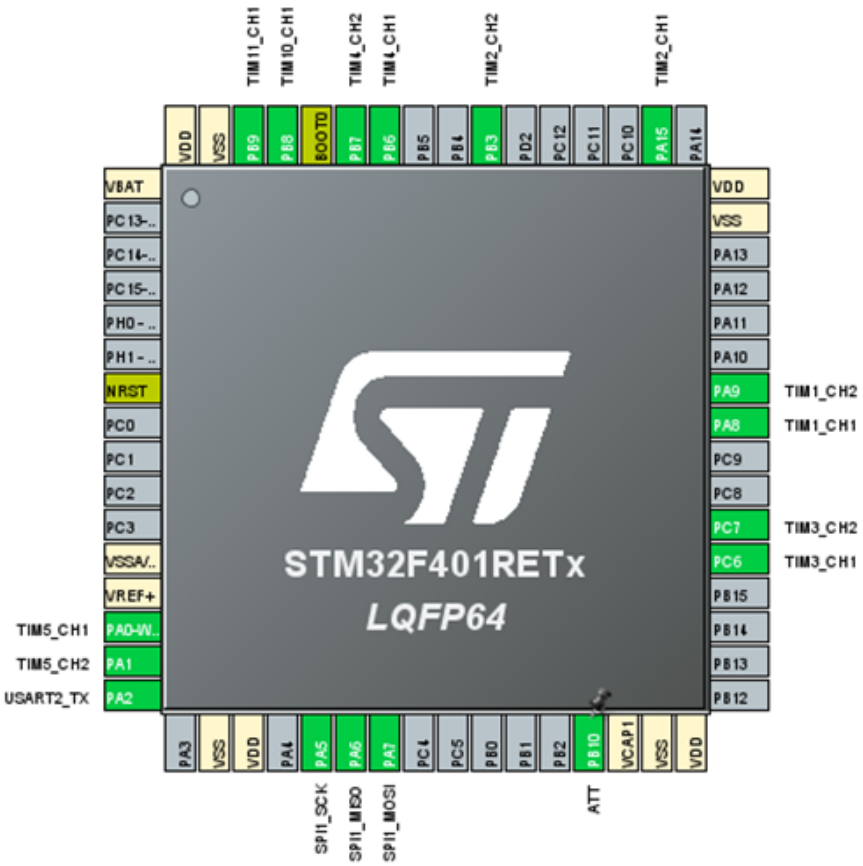


Figure 29: Project IOC made on CubeIDE

The SPI interface, used for the communication with the Lynxmotion controller, takes three pin on the board:

- PA5: propagate the clock signal;
- PA6: the MISO (Master Input Slave Output) channel, for receiving data;
- PA7: the MOSI (Master Output Slave Input) channel, for sending data.

However, the SPI protocol need another pin, acting as Slave Select: a signal set as high that, when a communication needs to be done, is set down to notify the slave about the incoming transmission. This pin, named ATT, is the PB10.
Lastly, a UART interface is set up for information logging purpose, such as the RPM and the computed PID output for each motor, and more, printed on a serial console (such as TeraTerm, PuTTY or the embedded console inside the IDE). It's configure is half-duplex mode, meaning that it is used only for transmissions, thus using only the pin PA2. However the pin is not connected to anything, since the communication make uses of the USB cable connected to the PC.

7.3 Clock tree setup

Before diving into the developed firmware, the last step is to setup the clock tree of the board. The clock tree is an interface that set the frequency of the main clock, propagated to various interfaces from which the actual timers get their working frequencies. The tree is visible in **Figure 30**, out of which can be seen the main clock (HCLK) set at $84MHz$. From here we need to set the APB2 Peripheral Clock, used by out SPI interface, to $21MHz$, and this will change the APB2 Timer Clock to $42MHz$.

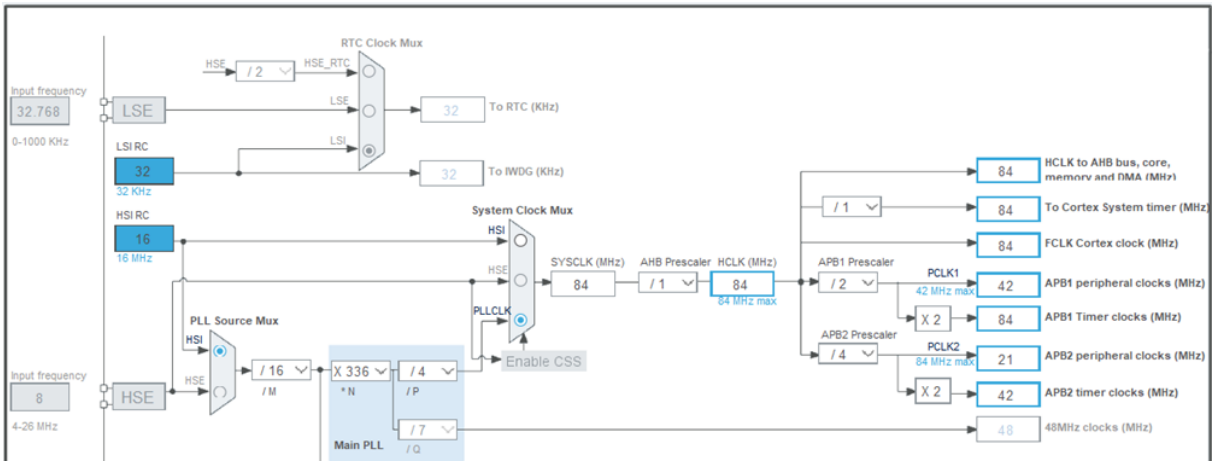


Figure 30: Clock Tree

With this last information, we need to compute the values of "Prescaler" and "Period" to set in the IOC for the TIM9, since this is the timer in charge of the callback function that need to be fired every $5ms$ (our sampling time). Computing the frequency as shown in:

$$f\ [\text{Hz}] = \frac{1}{T_s} = 200\ \text{Hz},$$

we can then compute the two values using the equation:

$$f\ [\text{Hz}] = \frac{\text{APB2 timer clock}}{(\text{prescaler} + 1) \cdot (\text{counter period} + 1)} \Rightarrow \begin{cases} \text{Prescaler} = 4199 \\ \text{Period} = 49 \end{cases}$$

In addition, we need to standardize the frequency output of all the PWM signals. From the STM32 datasheet we can notice that TIM10 and TIM11 uses the APB2, thus working at 42 MHz, while TIM2 uses the APB1, thus working at 84 MHz. Setting the prescaler values for all the timers, we can set the same working frequency for all the PWM signals. In particular the prescaler for TIM10 and TIM11 has been set at the value of 1049, while the prescaler for the TIM2 has been set at the value of 2049, thus achieving a working frequency of 40 kHz for all the timers.

8. Software implementation

Now that the IOC is completed, the firmware can be written. As already stated, the project make uses of the C code, and every part of the systems has it's own driver. In this chapter every aspect of the developed code will be explained.

8.1 Lynxmotion controller driver

The driver is composed of three function. The communication between the board and the BT controller is performed as a normal SPI communication: the CS (Chip select) channel, in the beginning, is set HIGH. When the master (the microcontroller) want to communicate with the slave (the controller), first it needs to set DOWN the CS channel. Then, a string of 9 bytes is sent, at which the PSX Controller reply with another string of nine bytes, representing the state of all the possible input:

- 0, digital input not pressed;
- 1, digital input pressed;
- 128, analogue stick in the neutral position;
- $[0, 127]$, analogue stick moved up/left;
- $[129, 255]$, analogue stick moved down/right.

The transmission buffer is a simple command to interrogate the controller on all states of its inputs. The communication is very fast, and the driver itself takes very little computational time. The structure implemented is as follow:

```
struct controller_s {
   uint8_t PSX_AD;
   uint8_t PSX_MOD;
   double speed;
};
typedef struct controller_s controller_t;
```

which implements:

- a variable to save the analogue stick input;
- a variable to save the controller mode (fast or slow) selected with the right and left pad buttons;
- a variable with the converted speed value.

The three implemented function are:

- **PSX_Init()**: function called in the initialization phase, such as:
  - set the analogue stick to the neutral value of 128, the speed value to 0, the MOD value (linked to the controller ID change) to 0, setting the fast controller, and the value of the X button to 0 (meaning not pressed);
  - set the CS Pin HIGH;

- **GET_PSX_DATA()**: using the HAL of the STM32, uses SPI functions to send and receive the data, using this specific operational pattern:
  - first the ATT pin is set LOW, to notify the controller of the incoming transmission, and wait 1 ms, to stabilise the communication;
  - then a specific HAL function, send and receive byte by byte (sending one byte and receiving one) for all the nine bytes in the transmission buffer;
  - lastly, after the transmit and receive are done, the ATT pin is set HIGH again, to notify the controller that the communication is finished, to ensure a proper transition.
- **DATA_DIGEST()**: read the RX buffer and interprets the received input, saving the data in the PSX struct; in particular:
  - the function scan the incoming buffer for specific position, assigned to the control we need to check the values; doing so, the software can ignore all other incoming states of buttons and illegal analogue sticks movements, that the user can do by mistake; the values of which we are interested are then saved in the appropriate struct;

The possible inputs are coming from the right analogue stick, the X button and the left and right pad button. Each of this input is linked to a specific functionality of the firmware:

- right analogue stick: controls the speed of the rover with the neutral position being stationary, the forward position represent a forward acceleration and the backward position represent a backward acceleration (both based on the inclination);
- right pad: set the PID coefficients to those of the slow controllers;
- left pad: set the PID coefficients to those of the fast controllers;

8.2 Encoder driver

The rotary encoder mounted on the electric motor send to the board two square wave signal, A and B, shifted by 90 degrees. By measuring the front occurrences of both waves the firmware can compute the actual RPM of the motor. In addition, by looking at which wave is "in front" of the other, the software can understand if the system is moving forward or backwards. The struct of the encoder is as follows:

```
struct encoder_s {
   TIM_HandleTypeDef* htim;
   uint8_t dir;
   uint16_t cnt;
   double last_tick;
   double current_tick;
   double speed;
};
typedef struct encoder_s encoder_t;
```

As visible from the code, the fields of the `encoder_t` structure are:

- pointer to a structure *TIM_HandleTypeDef*, which is the manager of the hardware timer associated with the encoder. The timer is used to count the ticks generated by the movement of the encoder and allows for proper handling of counts and readings in real time;
- a variable *dir* of type *uint8_t* that stores the direction of rotation where 0 indicates clockwise rotation and 1 indicates counterclockwise rotation;
- a 16-bit unsigned *cnt* counter that keeps track of the encoder's tick count. This field represents the total number of accumulated ticks, i.e. the angular distance travelled by the motor shaft;
- a variable *last_tick* of type *double* that stores the value of the previous tick count. It is used to calculate the variation of ticks between two instants of time, useful for determining speed;
- a variable *current_tick* of type *double* that stores the current tick count. It is used together with *last_tick* to calculate the tick delta;
- a floating point *speed* variable that contains the rotation speed of the encoder in RPM. This variable is calculated from the variation of the ticks and the sampling time.

The implemented functions are:

- **encoder_init()**: function called in the initialization phase, load all the parameters into the encoder struct, such as:
  - assign the specified timer to the specific encoder struct;
  - set all the parameters, inside the struct, to zero (such the direction value, the current and last tick, measured in ms, the speed and the encoder count value);
- **getTicksDelta()**: compute the ticks variation between two moments in time taking into account the overflow, if any; the computation starts with count of the fronts on the two square signals and the delta calculation (the variation) continue choosing between three cases:
  - if "$\delta = \text{currentTick} - \text{lastTick} \leq \lceil 6936^{1} \times T_s \rceil$" then the variation is equal to the simple difference between the two values;
  - if "$\delta = \text{currentTick} - \text{lastTick} > \lceil 6936 \times T_s \rceil$" and "lastTick > currentTick", then the computation need to account for an overflow with the following equation:
    $$\delta = currentTicks + 2^{16} - 1 - lastTicks;$$
  - if "$\delta = \text{currentTick} - \text{lastTick} > \lceil 6936 \times T_s \rceil$" and "lastTick < currentTick", the delta computation become:
    $$\delta = currentTicks - 2^{16} + 1 - lastTicks.$$

  This "δ" value is then returned and used for further calculations.
- **getSpeedByDelta()**: compute the speed in RPM, using the equation:
  $$Speed[RPM] = ticksDelta \cdot \frac{60}{48 \cdot 51 \cdot T_s},$$

  where:
  - "$ticksDelta$" is the output of the previous function;
  - 60, is the value of seconds in a minute;
  - "T_s", is the sampling time;
  - 48, represents the number of fronts to count for a complete revolution of the wheel;
  - 51, is the reduction gear constant.

All the data coming from the encoder driver are then saved into the four struct variables, one for each motor.

[1] This number represents the maximum number of ticks the encoder can count before resetting to zero.

8.3 PID controller driver

As for any other driver, the PID controller has a struct containing all of its parameters: the two coefficients, K_p and K_i, the setpoint, the newly calculated output (u) and the last computed output ($last\_u$), the error of the last cycle, the integral part and the controller ID (fast or slow). The coefficients needed to be discretized before using them into the firmware, using the following equation:

$$k_{pz} = k_p; \quad k_{iz} = k_i \cdot \frac{T_s}{2}; \quad k_{dz} = k_d \cdot \frac{2}{T_s}.$$

For example, the computation for extracting the front right PI controller coefficients proceed as follows:

$$C(s) = 0.054164 \cdot \frac{(s + 52.75)}{s} \rightarrow C(s) = 0.054164 + 2.857151\frac{1}{s}$$

$$k_p = 0.054164; \quad k_i = 2.857151; \quad k_d = 0.$$

This values can then be discretized using the equation seen before. Of course, the derivative coefficient K_d is zero, since that term is not present in the controllers. In **Table 4** are reported all the coefficients for the fast controllers, while in **Table 5** are reported all the coefficients for the slow controllers.

$PIDName$	K_{pz}	K_{iz}
Pid_RF	0.054164	0.0071
Pid_LF	0.059952	0.0067
Pid_RR	0.066368	0.0075
Pid_LR	0.056354	0.0069
Pid_FAx	0.059952	0.0067
Pid_RAx	0.056354	0.0069

Table 4: Fast controller coefficients

$PIDName$	K_{pz}	K_{iz}
Pid_RF	0.019625	0.002588
Pid_LF	0.021725	0.00244
Pid_RR	0.024046	0.002703
Pid_LR	0.020418	0.002516
Pid_FAx	0.021725	0.00244
Pid_RAx	0.020418	0.002516

Table 5: Slow controller coefficients

The structure used for the PID controllers is the following:

```
struct pid {
   double Kp;
   double Ki;
   double Kd;
   double prevError;
   double integral;
   double setpoint;
   double last_u;
   double u;
   uint8_t IDController;
};
typedef struct pid PID_t
```

where:

- K_p is the proportional coefficient;
- K_i is the integral coefficient;
- K_d is the derivative coefficient;
- $Prev\_err$, the error of the previous cycle;
- $integral$, the integral state;
- $setpoint$, the targhet value that the PID will try to reach;
- $last\_u$, the PID output of the last cycle;
- u, the actual PID output;
- $IDController$, the identificative number of the fast or slow controller.

The driver is composed of three functions:

- **PID_Init()**: the function set the parameters for the specific PI controller, associated to a motor, preparing the PID structure for use in a closed-loop control, ensuring that all variables are set correctly before starting the actual control:
  - all the coefficients, K_p and K_i are set to the right values, the values of the error is set to 0, as well as the value of the setpoint and those of the current and the previous output;
  - the controller ID, is set to 0, since at the start of the system, the fast controllers are loaded;
- **PID_Compute()**: compute all the necessary calculations (error, integrative term and current output), and save them into the struct, as such:
  - the error is computed as the difference between the setpoint (speed value to be achieved) and the actual speed value (computed by the encoder);
  - the proportional error is then computed as the multiplication between the proportional coefficient and the computed error;
  - the derivative part is then calculated multiplying the difference between the current and previous error and the derivative gain K_d;
  - the integral error is computed as the error multiplied by the integral gain K_i; this value is summed into the integral state if the stop summation control allows it;
  - the proportional and integral action are then summed and saved into the adequate variables in the controller struct;
  - lastly, a saturation check is applied, limiting the output in the range $[-12, 12]V$;

- **PWM_generate()**: normalize the current output in the range accepted by the PWM, and generate the PWM signal, such as:
  - a first saturation test, check if the output computed is in the max range value of $[-12, +12]V$, to avoid giving an output potentially disruptive for the system; if the variable is out of the specified range, it is limited to the right value (max positive or max negative);
  - subsequently the value computed in the range of the $\pm 12V$, is normalized to be fit in the range accepted by the motor driver, which we recall being $[1.875, 3.125]V$, using the equation:
$$\text{Voltage} = 2.5 + \text{Voltage} \cdot \frac{0.625}{12};$$
  - to generate the PWM wave, the firmware must convert this computed voltage value, into a "period" value, based on the maximum period of the timer, using the equation:
$$\text{val} = \frac{\text{Voltage}}{3.3} \cdot (\text{timerPeriod});$$
  - lastly, using the __HAL_TIM_SET_COMPARE function provided by STM drivers, the firmware can generate the PWM square wave, with the right ON and OFF timing.

A last function has been implemented that execute the switch behaviour:

- "**switchKcontrol()**" simply executes the manual controllers switch, when a change in the MOD value is detected:
  - switch the coefficients (K_p and K_i) and the IDController values for the controllers for each motor; to achieve a bumpless switch, an update of the integral values must be computed as:
$$\text{integral} = \text{integral} * \frac{\text{old\_}K_i}{\text{new\_}K_i};$$

Lastly, a temporary function called **rpm_to_voltage()** was created to map the RPM range into the tension range, to be used in testing the system without any control system: simply the input from the controller, will be scaled and given in output as a PWM wave, without going through the calculations for the control. To map the RPM to the tension, a coefficients has been computed as:

$$\text{conversion\_coefficient} = \frac{V_{MAX} - V_{MIN}}{RPM_{MAX} - RPM_{MIN}},$$

resulting in a coefficient of 0.075.
The function take as input the reference speed that we want to reach and return the current tension mapped in the range $[-12, 12]$. This tension is then passed as a parameters to the **PWM_generate()** function, that will generate the wave.

8.4 Logging driver

The logging, in general terms, is the process of collecting data, or events, from a running software. Reading a log can be extremely useful since:

- allows developers to track the behaviour of the program during execution, making possible to track errors, an unexpected software behaviour or to easily find what part of the code is being executed and what is the output computed; doing so, greatly speeds up the process of testing and, if necessary, bug-fixing;
- log can contain any data the developers want such as function output data, timers values, the duration of specific operations, the amount of memory used, CPU load or response times.

The logger implemented make uses of a circular buffer, a data structure in which the space is used in a cyclic mode: the end of the buffer is directly connected to the start, and the space is limited. The driver contains the struct and three functions:

- **cb_init()**: initialization function, allocate the right amount of memory necessary to the buffer, computed as:
$$\frac{2}{(T_s \cdot \text{prescaler})};$$
the *prescaler* values is set to 4, avoiding delays in the system that will make the rover not reactive enough. The sampling frequency drops from $\frac{1}{T_s}$ to $\frac{1}{4 \cdot T_s}$ which still satisfies the sampling theorem, since:
$$\frac{1}{4 \cdot T_s} = 50Hz$$
is still a greater value than:
$$2f_c = 2 \cdot \frac{\omega 3}{2 \cdot \pi}$$
which for the fast controlled system is equal to 12.70 Hz and for the slow controlled system is equal to 4.61 Hz, both less than 50 Hz.
- **cb_free()**: frees the memory associated with the buffer and resets all fields in the structure to NULL; it is used to clear resources when the buffer is no longer needed;
- **cb_push_back()**: add data to the buffer; change the value of the "writing" flag to TRUE, then add data to the head of the buffer, updating the head pointer value in the struct; at the end it change back the flag value to FALSE; if the buffer is full, raise an error;
- **cb_pop_front()**: copy an element from the buffer's tail, into another structure, and delete it from the main buffer; it waits if there are any writing process and, if the buffer is empty, raise an error.

The backing structure is called "record" and contains all the data, gathered from the buffer, that needs to be printed, for example on a console.

8.5 Callback function

The "Callback function" is the core of the system: it is execute every $T_s = 0,005$s, and is responsible for nearly all the computations in the firmware. The function, that calls most of the function just reported, is structured as it follows:

- gathers the actual speed, in RPM, from the encoders;
- then, calling the "**PID_Compute()**" functions, calculate all the output for the motors, including the axles outputs;
- calling the "**PWM_Generate()**" functions, generate and send the PWM signals to the GPIO pins on the board, which will propagate these signals into the RC filters, to the motor drivers;
- some data are then gathered, such as the PI values (error, current and last output) for every controller and the speeds computed by the encoders;
- the barrier values is changed to 1, releasing the firmware.

9. Validation on the real system

The validation process of the final system was performed in three steps:

- first, the behaviour of the system without controller has been tested;
- then only the single PI motor controllers were introduced, to check any improvement of the system;
- finally, the complete system has been loaded on the board, including the axles PI controllers.

Each step has been executed on standard acquisition, to check the exact same type of system behaviours in each validation. The acquisitions were made on the following inputs:

- full forward acceleration;
- slight forward acceleration.

Each acquisition shows from 5 to 13 values:

- on the first tests, only the RPM of the four motors has been acquired, for a total of five values;
- on the next second test, for each motor, it is show the PI controller computed output and the encoder computed RPM, for a total of nine values;
- in the last test, in addition to the motor values from before, for each axle it is show the PI controller computed output and the error computed on that specific axle, for a total of thirteen values.

The printed string is common between all the acquisition. The images shows, from left to right, the value for the motors, following the scheme:

RF: U,RPM - LF: U,RPM - RR: U,RPM - LR: U,RPM - FAx: U,e - RAx: U,e - Ref: Val

where "U" is the output computed by the PID, "RPM" is the speed measured and "e" is the error on the axle, computed as the difference between the speed of the left motor and the speed of the right motor. The Ref value report the reference speed sent to the system, by the PSX controller.

9.1 System without controllers

In this section the behaviour of the standalone system, without any control means, is being tested. This is an important test, necessary to validate the other two: having a reference of the system without controllers can improve the validation of further tests that make use of PID controllers.
The test followed the steps discussed before. In **Figure 30**, it is shown the behaviour of the system on a full forward acceleration, while in **Figure 31** is shown the acceleration behaviour, testing the transient state.

```
RF: 156.862745  LF: 161.764706  RR: 151.960784  LR: 161.764706   Ref: 160.000000
RF: 156.862745  LF: 156.862745  RR: 156.862745  LR: 156.862745   Ref: 160.000000
RF: 156.862745  LF: 156.862745  RR: 156.862745  LR: 151.960784   Ref: 160.000000
RF: 161.764706  LF: 161.764706  RR: 151.960784  LR: 156.862745   Ref: 160.000000
RF: 156.862745  LF: 156.862745  RR: 156.862745  LR: 156.862745   Ref: 160.000000
RF: 156.862745  LF: 161.764706  RR: 156.862745  LR: 156.862745   Ref: 160.000000
RF: 151.960784  LF: 161.764706  RR: 156.862745  LR: 151.960784   Ref: 160.000000
RF: 156.862745  LF: 161.764706  RR: 156.862745  LR: 156.862745   Ref: 160.000000
```

Figure 30: Steady state without controllers

```
RF: 0.000000  LF: 0.000000  RR: 0.000000  LR: 0.000000   Ref: 0.000000
RF: 0.000000  LF: 0.000000  RR: 0.000000  LR: 4.901961   Ref: 33.000000
RF: 19.607843  LF: 19.607843  RR: 19.607843  LR: 19.607843   Ref: 160.000000
RF: 53.921569  LF: 49.019608  RR: 63.725490  LR: 63.725490   Ref: 160.000000
RF: 112.745098  LF: 112.745098  RR: 107.843137  LR: 112.745098   Ref: 160.000000
RF: 137.254902  LF: 142.156863  RR: 137.254902  LR: 137.254902   Ref: 160.000000
RF: 147.058824  LF: 151.960784  RR: 147.058824  LR: 142.156863   Ref: 160.000000
RF: 151.960784  LF: 156.862745  RR: 151.960784  LR: 151.960784   Ref: 160.000000
RF: 156.862745  LF: 156.862745  RR: 156.862745  LR: 151.960784   Ref: 160.000000
RF: 151.960784  LF: 156.862745  RR: 151.960784  LR: 156.862745   Ref: 160.000000
RF: 151.960784  LF: 161.764706  RR: 156.862745  LR: 151.960784   Ref: 160.000000
```

Figure 31: Transient state behaviour without controllers

9.2 System with controlled motor

In this section the test performed on the system with only the four motor's PIDs will be reported

```
RF: 12.000000, 156.862745  LF: 11.425165, 161.764706  RR: 12.000000, 151.960784  LR: 12.000000, 156.862745  Ref: 160.000000
RF: 12.000000, 151.960784  LF: 11.737440, 156.862745  RR: 12.000000, 156.862745  LR: 12.000000, 156.862745  Ref: 160.000000
RF: 12.000000, 151.960784  LF: 11.722989, 156.862745  RR: 12.000000, 151.960784  LR: 12.000000, 156.862745  Ref: 160.000000
RF: 12.000000, 156.862745  LF: 11.414655, 161.764706  RR: 12.000000, 151.960784  LR: 12.000000, 156.862745  Ref: 160.000000
RF: 12.000000, 156.862745  LF: 11.400204, 161.764706  RR: 12.000000, 156.862745  LR: 12.000000, 156.862745  Ref: 160.000000
RF: 12.000000, 156.862745  LF: 11.418596, 161.764706  RR: 12.000000, 156.862745  LR: 12.000000, 156.862745  Ref: 160.000000
RF: 12.000000, 156.862745  LF: 11.404145, 161.764706  RR: 12.000000, 156.862745  LR: 12.000000, 156.862745  Ref: 160.000000
RF: 12.000000, 156.862745  LF: 11.389695, 161.764706  RR: 12.000000, 156.862745  LR: 12.000000, 156.862745  Ref: 160.000000
```

Figure 32: Steady state with only motor controllers

```
RF: 0.548510, 0.000000  LF: 0.340255, 0.000000  RR: 0.177941, 0.000000  LR: 0.154235, 0.000000  Ref: 0.000000
RF: 10.478550, 0.000000  LF: 11.125175, 0.000000  RR: 12.000000, 0.000000  LR: 10.399075, 0.000000  Ref: 160.000000
RF: 11.902079, 44.117647  LF: 12.000000, 44.117647  RR: 11.776547, 34.313725  LR: 11.472948, 34.313725  Ref: 160.000000
RF: 11.423922, 88.235294  LF: 10.951861, 83.333333  RR: 11.523880, 73.529412  LR: 11.081095, 78.431373  Ref: 160.000000
RF: 10.977001, 122.549020  LF: 9.867234, 122.549020  RR: 10.669743, 112.745098  LR: 10.789850, 112.745098  Ref: 160.000000
RF: 10.965648, 137.254902  LF: 10.151783, 132.352941  RR: 10.418409, 132.352941  LR: 10.617046, 132.352941  Ref: 160.000000
RF: 10.976177, 147.058824  LF: 10.500312, 137.254902  RR: 10.848958, 137.254902  LR: 11.002389, 137.254902  Ref: 160.000000
RF: 11.378511, 147.058824  LF: 10.717469, 142.156863  RR: 11.095684, 142.156863  LR: 10.908546, 147.058824  Ref: 160.000000
RF: 11.376118, 151.960784  LF: 10.803253, 147.058824  RR: 11.195351, 147.058824  LR: 10.955654, 151.960784  Ref: 160.000000
```

Figure 33: Transient state with only motor controllers

The slight forward acceleration proves that even in the transient state, the PI control increments the stability of the system: without control, as seen before, the acceleration had nearly all different RPM for all the motor, while with the PI controller at least the speed on the same side are consistent.

9.3 System with controlled motor and axle control

In this section, the tests performed on the complete system, four PI controller, one for each motor, and two PI controller, one for each axle, are showed.

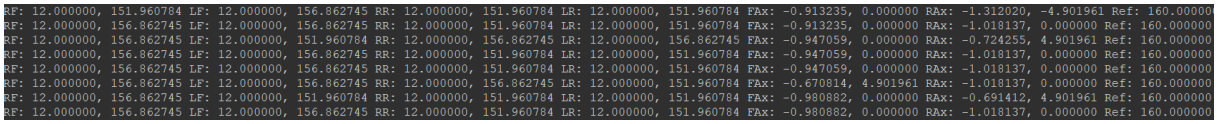


Figure 34: Steady state with axles controller

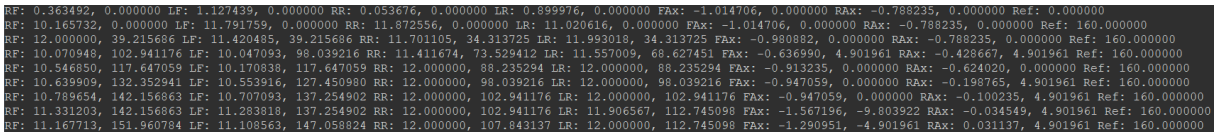


Figure 35: Transient state with axles controller

The system, even with slight RPM differences, is however steady and follow a mostly straight line. The only problem that make the rover slightly turn is an unbalancing in the alignment on one of the wheels. Apart from this, the system is well controlled by PI controllers.